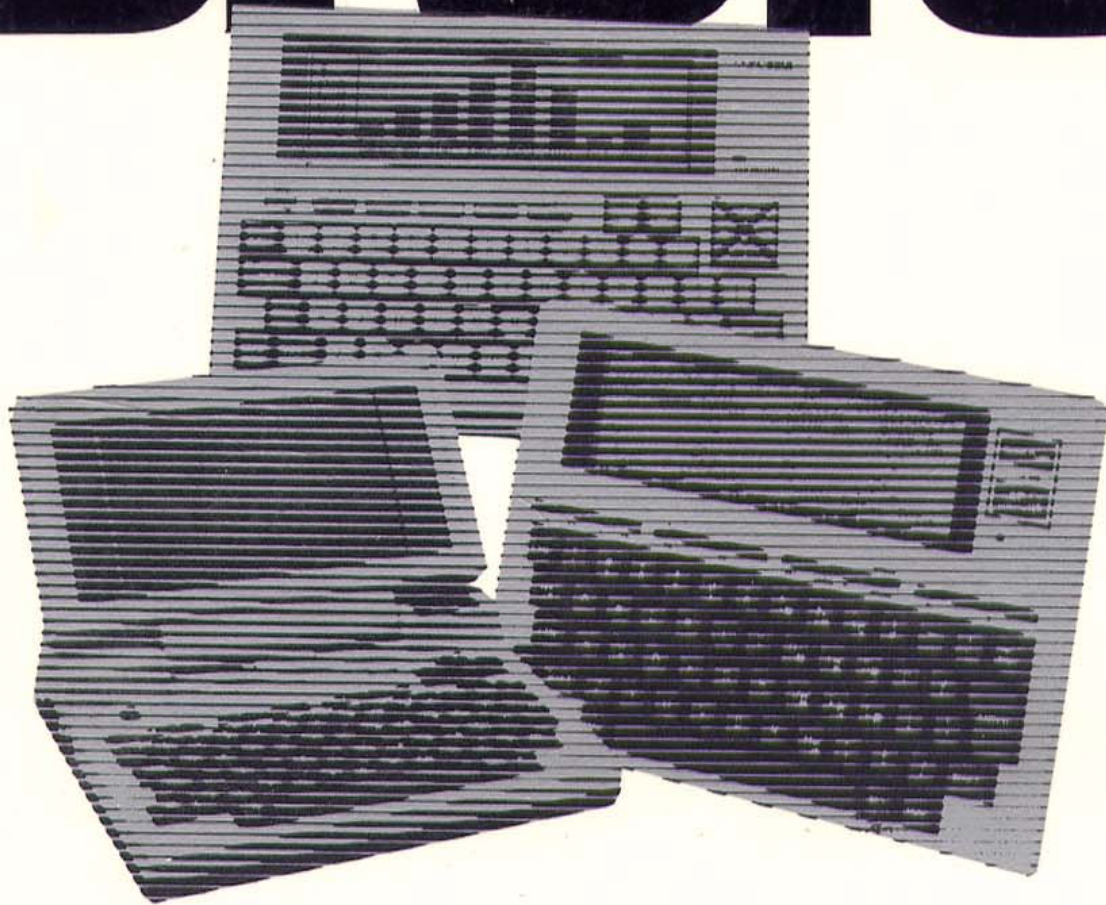


PORTABLE BASIC



**Programming the TRS-80 MODEL 200,
TRS-80 MODEL 100, and
NEC PC-8201A Computers**

DAVID THOMAS

PORTABLE BASIC



PORTABLE BASIC

P R O G R A M M I N G T H E
TRS-80 MODEL 200,
TRS-80 MODEL 100,
A N D
NEC PC-8201A
C O M P U T E R S

David Thomas

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá
Guatemala Hamburg Johannesburg Lisbon London
Madrid Mexico Montreal New Delhi Panama Paris
San Juan São Paulo Singapore Sydney Tokyo Toronto

The author of the programs provided with this book has carefully reviewed them to ensure their performance in accordance with the specifications described in the book. Neither the author nor McGraw-Hill, however, makes any warranties whatever concerning the programs. They assume no responsibility or liability of any kind for errors in the programs or for the consequences of any such errors.

NEC PC-8201A is a registered trademark of NEC Home Electronics.

TRS-80 is a registered trademark of Radio Shack, a division of Tandy Corporation.

PORTABLE BASIC: PROGRAMMING THE TRS-80 MODEL 200, TRS-80 MODEL 100, AND NEC PC-8201A COMPUTERS

Copyright © 1985 by David Thomas. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

A BYTE Book.

1 2 3 4 5 6 7 8 9 10 SEMSEM 8 9 3 2 1 0 9 8 7 6 5

ISBN 0-07-064260-5

LIBRARY OF CONGRESS CATALOGING IN PUBLICATION DATA

Thomas, David, (*date*)

Portable BASIC: programming the TRS-80 Model 200, TRS-80 Model 100, and NEC PC-8201A computers.

(A Byte book)

Includes index.

1. Basic (Computer program language) 2. TRS-80 Model 200 (Computer)—Programming. 3. TRS-80 Model 100 (Computer)—Programming. 4. NEC PC-8201A (Computer) —Programming. I. Title. II. Series: Byte books.
QA76.73.B3T46 1985 001.64'24 84-20113
ISBN 0-07-064260-5

Editor: Jeff McCartney

Editing Supervisor: Aliza Greenblatt

Book Design by Patrice Fodero

Contents

Preface	ix
Introduction	1
Lesson 1 Getting Started	8
Lesson 2 Numeric Constants and Variables	16
Lesson 3 Beginning Programming	21
Lesson 4 Strings—Alphanumeric Information	27
Lesson 5 EDITing Programs	31
Review Test 1	36
Lesson 6 INPUTing Information	38
Lesson 7 Program Branching—The GOTO Statement	43
Lesson 8 Decision Making—The IF THEN Statement	47
Lesson 9 Defining the BASIC Function Keys	53
Lesson 10 Using RAM Files	57
Review Test 2	62
Lesson 11 Order of Calculations	65
Lesson 12 Higher-Order Mathematical Functions	72

Lesson 13	The INT, FIX, and SGN Functions	75
Lesson 14	Numeric Variable Types	83
Lesson 15	Simulating Chance Occurrences	89
Review Test 3		97
Lesson 16	FOR TO NEXT Looping	99
Lesson 17	REM Statements	108
Lesson 18	Storing DATA in a Program	111
Lesson 19	Arrays and Subscripts	120
Lesson 20	The GOSUB and RETURN Statements	132
Review Test 4		141
Lesson 21	Using PRINT USING	144
Lesson 22	Multiple Statement Lines	151
Lesson 23	More on Decision Making	156
Lesson 24	More on FOR NEXT Looping	159
Lesson 25	Storing Programs on Cassette Tape	165
Review Test 5		170
Lesson 26	Adding SOUND to a Program	172
Lesson 27	String Comparisons	176
Lesson 28	String Manipulations	183
Lesson 29	The ASCII Code Functions	192
Lesson 30	Numeric/String Conversions	198
Review Test 6		201
Lesson 31	Reserving String Space	204
Lesson 32	LINE INPUT, INPUT\$, and INKEY\$	209
Lesson 33	Logical Operations	215
Lesson 34	The ON GOTO and ON GOSUB Statements	223
Lesson 35	LEFT\$, RIGHT\$, SPACE\$, and INSTR	229
Review Test 7		235

Lesson 36	Controlling the PRINT Position	237
Lesson 37	The Graphic Statements	251
Lesson 38	Data Files—Part 1	257
Lesson 39	Data Files—Part 2	265
Lesson 40	Some Final Instructions	273
Review Test 8		282
Appendix A	ASCII Codes and Characters	285
Appendix B	BASIC Reserved Words	299
Appendix C	Error Codes and Messages	301
Appendix D	Answers to Review Tests	304
Index		329



Preface

The objective of this book is to help you learn BASIC and to show you how to use that language to program your Tandy TRS-80 Model 200, 100, or NEC PC-8201A computer. This book assumes that you have the manual supplied with your computer and that you have read the discussions regarding the operation of the computer and the functions of the keys. The book does not assume that you understand everything in that manual, or that you have had previous programming experience. On the contrary, it assumes that you are now making your first efforts to learn how to program a computer.

If you have had previous programming experience, I believe you will still find this book useful. In addition to its many practical examples, this book describes capabilities of the computers that are not documented in the manuals supplied with the computers. For example, Model 100 owners will find in Lesson 15 a description of how to use a negative argument to make the RND function more random; in Lesson 36 a description of the action of the PRINT @ statement; and in Appendix A a description of 21 undocumented "escape" sequences that perform functions such as clearing the screen, clearing to the end of the screen, and positioning the cursor to any desired row and column location. PC-8201A owners will find in Lesson 37 a description of the **LINE** statement, a BASIC statement totally ignored by the manual supplied with the computer.

There are 40 lessons in this book. Each lesson covers an important programming concept or BASIC statement or command and is short enough to be read easily in one session. The lessons gradually build on the information discussed in the preceding lessons, so read the lessons in sequence. At the end of each lesson is a summary section. Use this section for a quick review and reinforcement of the contents of the lesson.

The only way to learn to program is to write programs and to run them. This book makes that easy. It contains example programs to illustrate every BASIC instruction discussed. The example programs, however, are designed to do more than just show how a statement or command works. They are also intended to arouse your interest and provide you with a core of useful programming routines that you can expand to suit your own needs. After reading the lessons and entering the given programs, try to write alternate versions. There are few things as flexible as programming, and there are many ways of making the computer perform the same actions. By composing programs of your own design, you will develop your programming skills more quickly.

You will find a review test after every fifth lesson. The questions in these tests have been selected to reinforce what you have learned in the preceding lessons and to challenge you to write programs using that information. Take the time to work these exercises. Complete answers to all questions are provided in Appendix D. The explanations accompanying many of the answers make additional points that were not specifically detailed in the text.

One of the remarkable things about computer programming is that you can learn to program from reading a book. The only requirements are enthusiasm, a desire to learn, and your own computer. As you learn the BASIC programming language, one of today's most useful tools, I trust that you will discover, as I have, that there are few activities more fascinating or rewarding than computer programming.

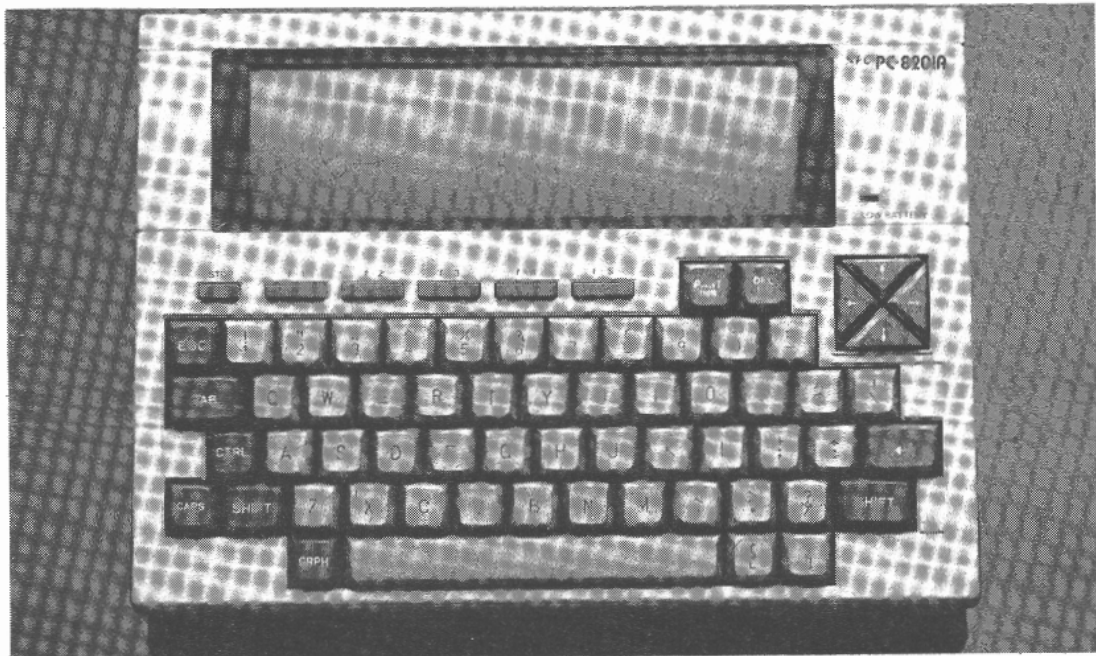
David Thomas

INTRODUCTION

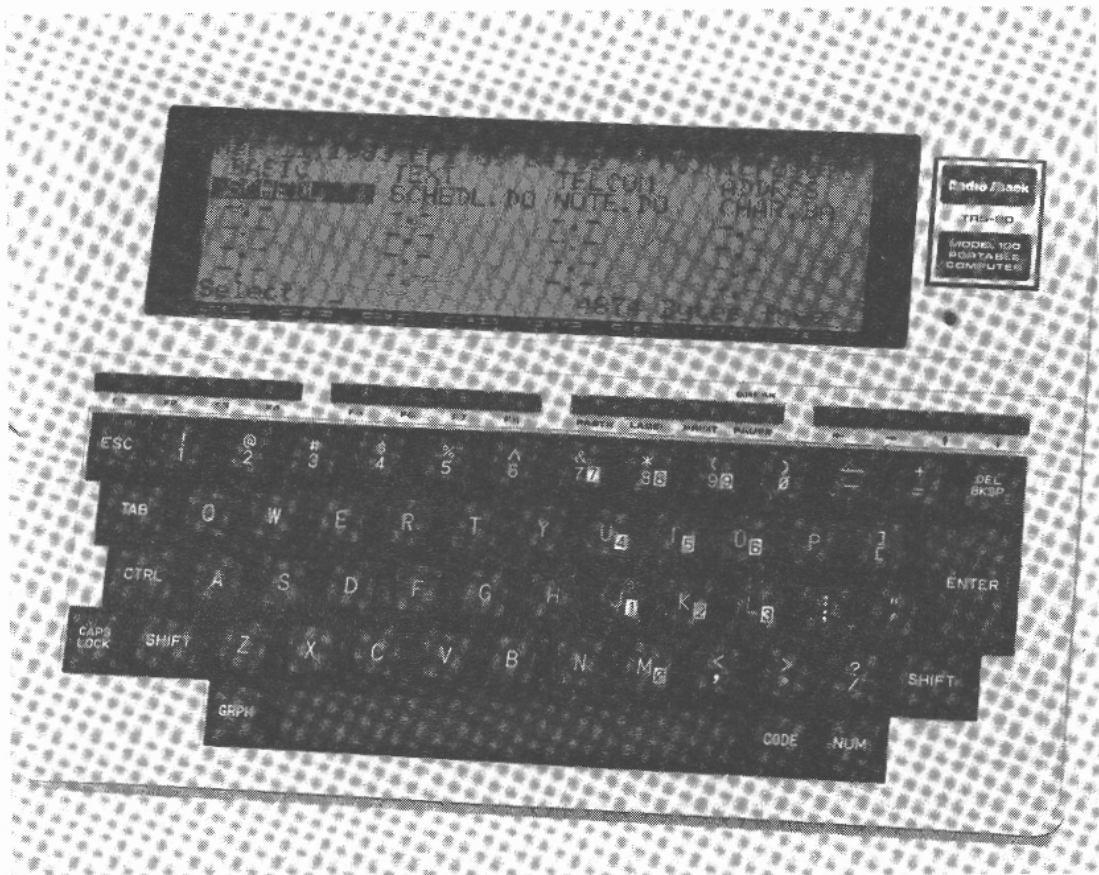
The Tandy TRS-80 Models 200, 100, and NEC PC-8201A computers mark the start of a new era in computing—full-featured portable computers that are both practical and affordable. The advent of truly portable computers, computers 4 or 5 pounds or less in weight and battery operated, means that this powerful tool can now be used anywhere. The usefulness of these computers, however, extends beyond their portability. With their innovative hardware features and powerful built-in software, the Model 200, Model 100, and PC-8201A can perform tasks that previously were achievable only by large, immobile “mainframe” computers.

The three computers, although very similar in many ways, have important differences. For example, the Model 200 has a display screen that tilts up and is twice as large as that provided on the Model 100 and PC-8201A computers. This introduction describes the physical characteristics of the three computers (see Fig. I-1).

2 / Introduction



(a) NEC PC-8201A



(b) TRS-80 Model 100

Fig. I-1 BASIC computers. (a: NEC Corporation; b,c: Tandy Corporation)



(c) TRS-80 Model 200

Fig. I-1 (*continued*).

The Model 100 and PC-8201A computers, and the Model 200 with its display screen folded, have almost identical physical dimensions. In length and width, they are about the size of a sheet of paper: $8\frac{1}{2} \times 11\frac{3}{4}$ inches. The machines differ only in the third dimension—the Model 100 is 2 inches high; the PC-8201A and the Model 200 are about $\frac{1}{2}$ inch higher.

All three computers have excellent keyboards—full-sized, typewriter-style, with a QWERTY arrangement of the alphabetic keys. The locations of the punctuation keys vary slightly, with the Model 200 and 100 adopting a typewriter arrangement more common in the United States and the PC-8201A using a layout that is more typically Japanese or European. The **ENTER** or carriage return key

is located in about the same place for the three machines, but the key is larger on the Tandy computers, probably making it easier to hit. On the PC-8201A, the enter key is marked .

In addition to the standard keys, there are special keys for computer functions. Here the machines also have differences. The Model 100 locates its cursor movement keys in a row above the typewriter keys. The Model 200 and PC-8201A computers have a more convenient design, placing their cursor keys in a "key-pad" arrangement; the Model 200 uses a "plus"-shaped arrangement, the PC-8201A uses diamond-shaped keys arranged in a square. Next to its cursor keys, the Model 100 has a row of four special keys: PASTE, LABEL, PRINT, and PAUSE/BREAK. The Model 200 has the same keys, but they are located in a row on the left side of the keyboard. The PC-8201A has physical equivalents for only three of these keys: INS/PAST, BS/DEL, and STOP.

The other major keyboard difference is in the function keys. The Model 200 and 100 have eight function keys; the PC-8201A has only five. But in spite of appearances, the PC-8201A has two more function keys than the Model 200 and 100 because the  key can be used with  through  to create five additional function keys. For example,  is obtained by pressing  . (The action of the function keys depends on the software program you are using.)

All keys automatically repeat if held down for approximately 2 seconds, and a keyboard buffer stores keystrokes that are entered too fast for the computer to act on them.

The display screen of the Model 200 is a tilt-up, 16-line by 40-character liquid-crystal display (LCD). The display screen of the Model 100 and PC-8201A computers is an 8-line by 40-character liquid crystal display. Thus the Model 200 has a display screen that is twice as large as the display screens of the Model 100 and PC-8201A computers. Although all three screens are small in comparison to the screens provided on most desktop computers, they have proven to be remarkably sufficient for portable applications. LCD screens are difficult to read in bad lighting. To help alleviate

this problem, the manufacturers include a dial on the left side of the machine for adjusting the display contrast.

In addition to their capability to display text, the LCD screens can display dot-addressable graphics. The Model 100 and PC-8201A screens supply a 240-column by 64-row graphic matrix. The Model 200 screen, because of its larger size, supplies a 240-column by 128-row graphic matrix. A fine feature of the screens is that the dot-addressable graphics can be mixed with text, providing many possibilities for graphs, charts, and games.

The heart of the computers is an 8-bit microprocessor known as an 8085. Because a CMOS (complementary metal-oxide semiconductor) version of the 8085 used, the power requirements of the computer are easily supplied by batteries. The four AA-size batteries used by the computers provide up to 20 hours of continuous operation. To prevent memory loss while changing batteries and when the computer is turned off, the manufacturers built in a rechargeable nickel-cadmium (nicad) battery. Additionally, the computers are equipped with an outlet for plugging in an optional ac adapter.

To prolong the life of its batteries, the computers automatically turn off after approximately 10 minutes of inactivity. The memory and display contents are retained after an automatic shutdown; simply turning the power switch OFF and ON restores the computer to its previous state.

A low-battery indicator lights when the power supply begins to approach the low end of its operational range, warning that only 5 to 20 minutes of battery power remain.

The three computers have different memory capabilities. The Model 100 can be expanded to a maximum of 32,768 (32K) bytes of random-access memory (RAM). (The official Tandy position is that the Model 100 can be expanded to only 24K of RAM. Third-party manufacturers, however, provide chips that can expand the RAM to the 32K limit mentioned above.) The Model 200 and PC-8201A computers can be expanded beyond this limit through the use of add-on RAM cartridges. The maximum RAM limit for the Model 200 is 72K. The maximum RAM limit for the PC-8201A is 64K. On both the

Model 200 and the PC-8201A, the additional RAM provided by the memory cartridges must be accessed in "banks."

Important to any computer are the input/output ports available for interfacing with peripheral devices such as printers, plotters, modems, cassette recorders, and bar-code readers. All three products have a full range of interfaces. Included in all three are a RS232 (serial) interface, Centronics (parallel) printer interface, cassette recorder interface, and bar-code reader interface. The three computers also have bus-expansion ports for connecting devices such as diskette drives or CRT (video) terminals. The Model 200 and PC-8201A have several additional ports, including ports for plugging in the optional add-on RAM cartridges.

One of the primary uses of a portable computer is telecommunications (communicating with other computers over the telephone line). Both Tandy computers have built-in, internal modems, so no additional hardware is required for connecting the computers to a telephone line. The PC-8201A does not have a built-in modem, but NEC manufactures an external modem that connects to the computer and permits telecommunications.

Sound capabilities are provided for each computer by the inclusion of a piezoelectric tone generator. The volume of the tone generator is fixed, but the pitch ranges through more than five octaves. A built-in electronic clock enables the computers to maintain the current time and date setting.

In addition to many hardware innovations, the computers' manufacturers made the bold decision to include built-in software programs. The Model 200 has five application programs permanently stored in its read-only memory (ROM): TEXT, TELCOM, ADDRSS, SCHEDL, and MSPLAN. The Model 100 has the TEXT, TELECOM, ADDRSS, and SCHEDL programs stored in its read-only memory. The PC-8201A has TEXT and TELCOM permanently stored in its ROM.

The TEXT program is a well-designed text editing program that is easy to learn and powerful enough to satisfy most writer's needs. It has commands for copying and moving text, searching for particu-

lar words, storing a text file on cassette tape (or other computer devices), and printing. The text files created by TEXT are automatically saved in RAM.

The TELCOM program, in conjunction with a modem, enables the computers to communicate with other appropriately programmed computers throughout the world over regular telephone lines. TELCOM provides commands for dialing phone numbers, transmitting passwords and log-on sequences, and sending or receiving text files. The portability of the computer makes it ideal for use as a remote terminal from your home, office, or motel room.

The ADDRSS and SCHEDL programs, available on the Model 200 and 100 computers, search files created by the TEXT program for address and schedule information. The data stored in the address and schedule files must follow a predefined format.

The Model 200 has a version of Microsoft's MultiPlan (MSPLAN) built into its ROM. MSPLAN has the same command structure as other versions of MultiPlan, and permits creation of spreadsheets as large as 63 columns by 99 rows. Besides MSPLAN, the Model 200 has several other unique software features, including alarm and calculator programs.

The version of BASIC supplied with the computers was designed by Microsoft and is a full-featured BASIC that is more powerful than the BASIC provided on many desktop computers. In addition to programming instructions normally provided only with diskette-based versions of Microsoft BASIC, the manufacturers include instructions designed specifically to take full advantage of the many hardware features of the computers.

As you might expect, there are minor differences in the BASIC languages. But those differences, as well as your course in BASIC programming, are the subject of this book. To step into the wonderful world of portable BASIC, turn the page and begin reading.

L E S S O N

1

Getting Started

If you have not done so already, set up your computer as described in the manuals supplied with the computer. Then turn it on. The display should show the initial Menu shown in Figs. 1-1, 1-2, and 1-3 (your Menu Screen may vary slightly, depending on the number of files stored in the computer).

Select the BASIC language interpreter from the Menu Screen by pressing the **ENTER** () key. (The computer powers up with the shaded cursor used to select menu options located over the word BASIC. If you have moved the cursor since turning on the computer, direct it to the word BASIC before pressing **ENTER**.)

The display will clear and the Microsoft copyright notice will appear. The display will also identify the make of computer (TRS-80 or NEC) and the number of bytes of free (unused) memory. Following these messages, the display will show the letters **Ok** and a flashing cursor

```

1984/11/19 11:24:26      (C) Microsoft #1
BASIC      TEXT      TELCOM      CHAP4.DO
CHAP5.DO   TEST2.DO   IOTA.BA   YesNo.BA
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
Load      Save      Name      List      28580

```

Fig. 1-1 Menu of the NEC PC-8201A computer.

```

Nov 19,1985 Mon 11:24:25  (C)Microsoft
BASIC      TEXT      TELCOM      ADDR55
SCHEDL     INTRO.DO  CHAP1.DO  CHAP2.DO
CHAP3.DO   TEST1.DO  SORT.BA   OMEGA.BA
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
Select: -      8580 Bytes free

```

Fig. 1-2 Menu of the TRS-80 Model 100 computer.

```

Apr 01,1985 Mon 11:24:25 #1 (C)Microsoft
BASIC      TEXT      TELCOM      ADDR55
SCHEDL     MSPLAN    MESON.DO  NANCY.DO
WD.BA      PEGGY.DO  MARY.DO  ASSETS.CO
QUARK.DO   GLUON.DO  -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
.-.-      -.-      -.-      -.-
18097 Bytes free
Bank      Copy Kill

```

Fig. 1-3 Menu of the TRS-80 Model 200 computer.

block. The `Ok` and the flashing cursor indicate that the computer is ready to accept your instructions. Now that the computer is awaiting your direction, try typing `THIS IS A GREAT COMPUTER` to get the feel of the keyboard. Notice as you type that the cursor moves to indicate where the next typed letter will appear on the screen. If you make a typing mistake, press the `BKSP` (`BS` on PC-8201A) key to move the cursor backward and erase the previous character, so that you can retype the correct letter. If you hold down the `BKSP` (`BS`) key, the cursor will repeatedly move left until it reaches the left edge of the display, or until you release the key.

When you have correctly typed `THIS IS A GREAT COMPUTER`, press the `ENTER` key. The function of the `ENTER` key is to instruct the computer to act on what you have typed in. In this case, the computer responds by displaying the message `?SN Error`, and then redisplaying `Ok` and the cursor. The `?SN Error` message appears because you did not type a valid BASIC instruction, and the computer responds only to instructions that it understands.

As you work through this book, you will learn what instructions the computer understands. If you misspell a BASIC instruction or enter an invalid instruction, the computer will respond with an *error message* such as the one you now see. Don't worry when you see an error message. These messages are the computer's method of indicating that it simply does not understand your instructions.

To illustrate an instruction that the computer does understand, type `PRINT 5+6` and press `ENTER`. (Use the `BKSP` (`BS`) key to correct any typing mistakes before you press `ENTER`.)

The computer will display the result of `5+6`, as shown below.

```
PRINT 5+6
  11
Ok
```

The word **PRINT** instructs the computer to display information on the screen. In this example, **PRINT 5+6** told the computer to display the result of adding 5 and 6. Notice that a "character space" was typed between **PRINT** and `5+6`. This space is not required for

the computer to understand the instruction. The space could be omitted, or additional spaces could be inserted, and the computer would still perform the instruction correctly.

```
PRINT5+6
11
OK
PRINT          5+6
11
Ok
```

Different computer manufacturers provide BASICs that vary enormously in their requirements for spaces. That the TRS-80 Models 200 and 100, and the NEC PC-8201A permit you to omit all spaces if you desire is an important advantage in these computers, because it permits programs to take up less memory than they would require if they *had to* include spaces. Unless memory size is critical, however, it is usually preferable to include spaces—they make computer instructions easier for human beings to understand.

The use of uppercase or lowercase letters in your instructions to the computer is also optional.

```
print 5+6
11
Ok
```

You can even intermix uppercase and lowercase letters without confusing the computer.

```
pRiNt 5+6
11
Ok
```

As an alternative to the **PRINT** statement, you can use the question mark (?) to instruct the computer to display information. For example, type ? 5/6 and press **ENTER**. The result is shown below.

```
? 5/6
.8333333333333333
Ok
```

12 / Lesson 1

The slash $\boxed{/}$ key is used to perform division. Therefore, the sequence ? 5/6 instructs the computer to display the result of 5 divided by 6.

A **PRINT** or ? sign is needed whenever you want the computer to display information on the screen. The two instructions are identical in their operation and can be used interchangeably. To familiarize yourself with the function of these instructions, work the problems below. These examples illustrate how you can use your computer to perform calculations.

Problem	BASIC Key Sequence
$121 + 122 + 123$	PRINT 121+122+123
$7.123 - 9$	?7.123-9
$629 \times 5.2 + 92 \times 2.5$	?629*5.2+92*2.5
$3 \div 17 + 9 \times 5 - 51$	PRINT 3/17+9*5-51

Note that the asterisk $\boxed{*}$ key is used to perform multiplication.

After working these problems, the screen should appear as shown below. (The results of the first two problems have shifted off the top of the screen, a process known as *scrolling*.)

```
Ok
?629*5.2+92*2.5
 3500.8
Ok
PRINT 3/17+9*5-51
-5.823529411765
Ok
```

Look closely at the answers displayed by the computer. Notice that the positive numbers do not line up with the **PRINT** and ? instructions, but the negative numbers do. This apparent inconsistency occurs because BASIC automatically places a character space in front of positive numbers. This space corresponds to the position occupied by the minus sign of a negative number and is added so that the digits of both negative and positive numbers begin in

the same relative location. Since this space is placed in front of the number, it is referred to as a *leading space*.

In addition to adding a leading space before positive numbers, BASIC also adds a space after every number. This *trailing space* is intended to separate the number from any information that may be displayed following the number.

One of the features of the **PRINT** and ? instructions is that they permit you to display more than one piece of information at a time. For example, type

```
PRINT 100;55
```

and press **ENTER**. The result is shown below.

```
PRINT 100;55
100 55
Ok
```

The computer displays the two values on the same line, separating them only by the leading and trailing spaces that it displays with the numbers.

When multiple values are displayed using the **PRINT** or ? instructions, the values are known as an *expression list*. In the example shown above, a semicolon separates the two values being displayed. You can also use a comma to separate print values:

```
PRINT 100,55
100          50
Ok
```

The comma instructs the computer to display the values at certain column locations known as *print zones*. There are two print zones per line, corresponding to the first and fifteenth character positions on the line. If more values are **PRINT**ed than can be displayed on one line, the computer advances to the corresponding zones on the next line.

```
PRINT 1,2,3,4,5,6,7
1           2
3           4
5           6
7
Ok
```

Semicolon and comma separators can be mixed as desired in an *expression list*:

```
PRINT 1;100,2;55
1  100      2  55
Ok
```

Here is another BASIC instruction that you will find useful: **CLS**. The **CLS** statement clears the display and positions the cursor to the upper left corner of the screen. To execute the statement, type **CLS** and press **ENTER**.

This concludes the first lesson. To leave BASIC and return to the Menu Screen, type **MENU** and press **ENTER**. As an alternative, you can press the **F8** function key on the TRS-80 Models 200 and 100, or the **F-10** function key on the NEC PC-8201A (the **F-10** key is accessed by pressing **SHIFT F-5**). *Note:* The function keys can be assigned different definitions in BASIC, as described in Lesson 9. If you have changed the definition of this key, it will not perform the **MENU** command, and you must type **MENU** and press **ENTER** to leave BASIC.

SUMMARY

The BASIC language interpreter is selected by pressing **ENTER** when the cursor is located over the word BASIC on the Menu Screen. Once in BASIC, the computer signals its readiness to accept instructions by displaying the word **Ok** and a flashing cursor.

Instructions to the computer can be typed in either uppercase or lowercase letters. Spaces between BASIC words used to instruct the computer can either be omitted or included. Omitting the spaces

reduces the amount of memory needed to store the instructions, but makes them more difficult for human beings to understand.

The word **PRINT** and the question mark (?) instruct the computer to display information on the screen. One of the features of these instructions is that they permit you to display more than one piece of information at a time. If multiple items are displayed (an *expression list*), they must be separated by semicolons or commas. The way in which the information is displayed depends on the punctuation selected:

- When a semicolon is used between **PRINTed** items, the computer displays the information adjacently. Remember, however, that BASIC automatically adds a space after every number and before each positive number.
- When a comma separator is used, the computer displays the information at certain column locations known as *print zones*. These zones are 15 columns apart and correspond to the first and fifteenth character positions on the line.

The BASIC instruction for clearing the display screen is **CLS**. The instruction for leaving BASIC and returning to the Menu Screen is **MENU**.

Numeric Constants and Variables

In Lesson 1 you instructed the computer to display numbers and the results of calculations by using statements such as

PRINT 6

and

PRINT 23-7

Numbers such as those illustrated above are known as numeric *constants*, since their value cannot vary. In programming, you will more often use numeric *variables* to represent numbers. A numeric variable is a "name" assigned a number value that can be used in place of a numeric constant.

Variables are assigned number values in BASIC using the equals (=) sign. For example, type

X=14

into the computer. Before pressing **ENTER**, consider this sequence carefully. It illustrates an important difference between the use of the equals sign in BASIC and the use of the equals sign in standard mathematics.

In mathematics, the equals sign indicates that two values or expressions are equal. For example, $2 + 3 = 5$ is a familiar mathematical equation indicating that $2 + 3$ and 5 are equal (the same). The equals sign in this usage simply states a fact—that the two sides of the equation are equal.

In BASIC, the equals sign performs an *action*: it defines or changes the value of the variable on the left side of the equals sign. In the example that you just typed in, the equals sign *assigns* the value 14 to the variable name "X". That variable may have had no previous value, or it may have had another value. In any case, after the execution of this sequence, X will have a value of 14.

Now press **ENTER** to execute the $X=14$ assignment sequence. The computer displays **Ok**, indicating that it has performed the assignment operation. That X now has a value of 14 can be easily checked. Simply instruct the computer to display the value of X by typing **PRINT X** and pressing **ENTER**. The result is shown below.

```
X=14
Ok
PRINT X
  14
Ok
```

As the word "variable" implies, the value of a variable can be changed simply by assigning it a new value using the equals sign. This flexibility makes variables extremely useful in programming.

Type

```
X=X+X
```

and press **ENTER**. Then perform the check again, as shown below.

```
X=X+X
Ok
PRINT X
28
Ok
```

The new value of X is 28. It was obtained by assigning X the value of $X + X$. Notice that the “old” value of X was used in assigning a new value to X. A BASIC sequence such as $X = X + X$ cannot be confused with a mathematical sequence, since it makes no sense as an equation; it would be like saying that $1 = 1 + 1$.

When assigning a value to a variable, always place the variable name on the left side of the equals sign and the value on the right side. This is a fundamental rule of BASIC programming.

Technical Note: Some BASICs require that assignment sequences be preceded by the BASIC statement **LET** (for example, **LET X=14** or **LET X=X+X**). Your computer’s BASIC leaves this choice to you. You can use or not use **LET** as desired. However, since dropping the **LET** reduces your typing effort and makes no difference in the computer’s operation, it is not used in this book.

The initial value of any numeric variable is zero. This is a convention of BASIC that you will find useful and convenient. Once a variable has been assigned a nonzero value, the variable retains that value until altered by another BASIC instruction or the computer is instructed to leave BASIC and return to the Menu Screen.

The BASIC command **NEW** also sets the value of all variables to zero, as well as erasing any program currently in the BASIC work area. The primary advantage of this command is that it allows you quickly to prepare the computer for a new program by clearing all information, including the values of variables, from the BASIC workspace.

In your computer’s BASIC, a variable name can be almost any sequence of letters and digits, as long as the name begins with a letter. For example,

X
PAYROLL
COUNT12
A7C3
PAGE

are all valid variable names. But

3X
B12?
INTEREST-COSTS
YRDS/MI

are invalid names because they either begin with a number or contain characters that are not valid (in these examples, the question mark, hyphen, or slash, respectively).

Only the first two characters of a variable's name are considered by the computer, however. Therefore, although names longer than two characters have the important advantage of clearly suggesting the use of the variable, the characters in the name beyond the second do not help differentiate the variable name. For example, to the computer, PAGE and PAYROLL represent the same variable name, PA.

Your computer puts one other restriction on variable names—they cannot contain any of the words used by BASIC for program instructions. For example, "**LET** = 5" is invalid because **LET** is part of the BASIC language of the computer. PRUNE is another example of an invalid variable name—it contains the BASIC word **RUN**. A list of the words used by BASIC, known as *reserved words* or *keywords*, is given in Appendix B.

SUMMARY

Numbers entered into the computer can be of two types: *constants* and *variables*. Specific values such as 3.313 are referred to as constants. "Names" assigned to numbers are called variables.

Variable names are formed by applying the following rules:

1. The first character in the name must be a letter of the alphabet, but that initial letter can be followed by one or more digits or letters.
2. The computer considers only the first two characters of the name, no matter how long the name.
3. Variable names cannot contain BASIC reserved words. The reserved word list is given in Appendix B.

The equals sign is used to assign values to variables. The variable name must always be on the left side and the value on the right side of the equals sign.

Until assigned another value, all numeric variables are equal to zero. Once assigned a nonzero value, a numeric variable retains that value in memory until it is replaced or cleared. The value of a variable can be replaced by assigning it a new value. It can be cleared by executing a **NEW** command or by returning to the Menu Screen. The **NEW** command instructs the computer to set all numeric variables to zero and erase any program currently in the BASIC workspace.

Beginning Programming

Lessons 1 and 2 illustrated using the computer in what is known as *immediate execution* or *command* mode. Such actions require the correct use of BASIC commands and statements, but they are not “programming.” *Programming* consists of storing a set of instructions in the computer’s memory. The stored instructions, or program, are not executed when you enter them, only after you command the computer to **RUN** the program.

To store an instruction in memory, you must precede the instruction with a *line number*. A line number directs the computer to remember, but not execute, the BASIC instructions that follow.

Clear the BASIC work area and set the value of all numeric variables to zero by typing **NEW** and pressing **ENTER**. Then type the following sequence exactly as shown:

```
100 GRADE=87
```

Before pressing **ENTER**, check that you have typed the example correctly. If not, use the **BKSP** key to correct it. When the sequence is correct, press **ENTER**. The computer will store the "line" in memory. (If you press **ENTER** before you see that you have made a mistake, simply retype the line correctly and press **ENTER**.)

To confirm that the assignment statement has not yet been executed, **PRINT** the value of **GRADE**.

```
100 GRADE=87
PRINT GRADE
0
Ok
```

Now execute the program by typing **RUN** and pressing **ENTER**. The computer responds by displaying **Ok**, indicating that it has executed the program. Check this by displaying the value of **GRADE** again. It now has a value of 87.

```
PRINT GRADE
87
Ok
```

This program, although very short, clearly illustrates the difference between command mode and programming. In command code, instructions are executed when **ENTER** is pressed. In programming, execution is deferred until the program is **RUN**.

The distinction between command mode and programming is also the basis for separating BASIC words into commands and statements. *Commands* are BASIC words that originally could be executed only in command mode. *Statements* are BASIC words that could be executed either in command mode or in a program. Although the traditional terminology is still used, your BASIC no longer restricts commands to command mode. Like statements, they can now be executed either in command mode or in a program.

The one-line program that you have just executed can be added to and made more useful by entering more program lines. For example, enter

```
110 PRINT GRADE
```

Notice that this line has a larger line number than the first. The computer uses the size (or magnitude) of the line numbers that you enter to determine the order in which program lines are executed. Since 110 is larger than 100, line 110 will be executed after line 100 when you run the program.

RUN the program again.

```
RUN
87
Ok
```

The program works as expected, assigning and displaying the value of GRADE.

Technical Note: Some BASICs require the use of an **END** statement at the end of a program. With your computer, this statement is optional when the end of the program is the last line of the program. It is possible, however, to design a program to end at a line number other than the last. When this occurs, an **END** statement is no longer optional. In this book, **END** statements are omitted except where required, as explained above. See Lesson 20 for an example of a program that requires an **END** statement.

This program illustrates one of the main advantages of programming your instructions over executing them in command mode: they can easily be executed more than once. To re-execute these instructions, **RUN** the program again. The program will remain in memory until you clear it by executing a **NEW** command.

As you work through this book, you will write programs that are longer and more complicated. When a program grows beyond 10 to 15 lines, it becomes difficult to remember what instructions you have already entered. Therefore, BASIC provides a command to allow you to view or **LIST** program instructions.

The **LIST** command has two forms: **LIST** and **LIST line-number range**. When **LIST** is executed without a *line-number range*, the

computer displays the entire program, beginning with the first line in the program.

The sole difference in executing **LIST** with a *line-number-range* parameter is that the listing begins with the line number specified by the *line-number range*. A *line-number range* can be either a single line number or a range of line numbers specified by a beginning line number, a hyphen, and an ending line number. If only a beginning line number and a hyphen are entered, the ending line number is assumed to be the last line in the program.

To illustrate the **LIST** command, **LIST** the grade program.

```
LIST
100 GRADE=87
110 PRINT GRADE
OK
```

Before ending this lesson, it is important to cover several additional points regarding BASIC line numbers. First, observe that we began this program with line number 100 and incremented that number by 10 for the next line. The actual numbers selected were unimportant. A functionally equivalent program could have been written using any two line numbers in the valid range 0 through 65529, *as long as the lines remained in the same relative order*.

Line numbers should not normally follow each other in increments of 1, however. By selecting line numbers with unused values or "gaps" between them, you make it easy to insert additional program lines. Simply enter the new line with a line number between the existing line numbers.

For example, suppose that you want to increase the value of grade by 7. This is accomplished by entering the following line:

```
105 GRADE=GRADE+7
```

Confirm that the line has been inserted between lines 100 and 110 by listing the program again.


```
LIST
100 GRADE=14
105 GRADE=GRADE+7
110 PRINT GRADE
Ok
```

Another important point regarding line numbers is the ease with which they allow program instructions to be replaced or deleted. To replace a program line, simply enter a new line with the same line number. Since BASIC does not allow a program to have two lines with identical line numbers, the computer automatically replaces the original with the new line. Test this feature by replacing line 105 with

```
105 GRADE=GRADE+11
```

LIST the program again.

```
LIST
100 GRADE=87
105 GRADE=GRADE+11
110 PRINT GRADE
```

The original line has been replaced with the new one.

Program lines are deleted by simply entering the number of the line that you want to delete. You can test this feature by entering

```
105
```

If you now enter **LIST**, no line 105 will be displayed.

```
LIST
100 GRADE=87
110 PRINT GRADE
```

SUMMARY

Every BASIC program, from the simplest to the most complicated, consists of one or more numbered "lines" of BASIC instructions.

The numbers that precede each line serve three purposes:

1. They allow the computer to distinguish between instructions to be stored in memory (a program) and instructions to be executed immediately (command mode).
2. They determine the order in which program instructions are executed.
3. They provide a method of identifying program instructions that you can use when listing or editing a program.

Line numbers must be whole numbers in the range 0 through 65529. As a rule, select line numbers that are 10 or more units apart. This practice makes it easy to insert additional lines.

The **LIST** command is used to display program lines. The command can be used singularly or with a *line-number-range* parameter. When **LIST** is used by itself, the listing begins with the lowest-numbered program line. When used with a *line-number-range* parameter, the listing begins and ends with a specified line number.

To delete a line, enter the number of the program line that you want to delete. If you attempt to delete a line number that does not exist, the computer will display a ?UL Error (undefined line) message.

Strings— Alphanumeric Information

The most significant feature of a computer is not the capability to manipulate numbers, since a pocket calculator can do that, but its additional capability to manipulate letters, symbols, signs, words, sentences, and in general, textual information of any type. To enable BASIC to distinguish between textual information and numbers, *textual information must be entered enclosed in quotation marks*. Such information, which can consist of any collection of alphabetic letters, signs, symbols, or digits from 0 to 255 characters in length, is referred to as a *string*. For example, enter

```
PRINT "This is a sample string"  
This is a sample string  
Ok
```

Observe that the computer displays exactly what you enter between the quotes, but not the quotes.

As you might suspect, a string such as that illustrated above is referred to as a *string constant*. Like a numeric constant, it can be changed only by retyping. You can create *string variables*, however, by simply assigning a string to a variable name that has as its *last* character a dollar sign (\$).

```
ITEM$="Television"
Ok
PRINT ITEM$
Television
Ok
```

If you attempt to assign a string to a variable name that does not end in a dollar sign, you will get a ?TM Error (Type Mismatch) message. Other than this requirement, the rules governing the creation of string variables are the same as those for numeric variables: they must begin with a letter; only the first two characters of the name are significant; and they cannot contain words on the reserved word list.

As you would expect, string constants and variables can be displayed in an *expression list*, and can be intermixed with numeric constants and variables.

```
TYPE$="Black and White"
Ok
PRINT TYPE$;ITEM$
Black and WhiteTelevision
Ok
PRINT 1;TYPE$;ITEM$
1 Black and WhiteTelevision
Ok
```

Notice that the computer does not automatically add spaces before and after strings as it does for numbers. If you want spaces displayed between the strings, you can include spaces in the appropriate places within the strings:

```
TYPE$="Black and White"
Ok
```

```
PRINT 1;TYPE$;ITEM$
1 Black and White Television
Ok
```

Or set a string equal to a single space and display it or a space constant between print items.

```
TYPE$="Color"
Ok
SP$=" "
Ok
PRINT 1;TYPE$;SP$;ITEM$
1 Color Television
Ok
PRINT 1;TYPE$;" ";ITEM$
1 Color Television
Ok
```

As stated above, a string can be from 0 to 255 characters in length. That a string has a maximum length is logical, but you may well wonder how a string can consist of zero characters. Such a string is referred to as a *null* string and is created by the sequence "" (there is no space between the quotes). (Lesson 27 illustrates several applications of the null string.)

A string, even one consisting of all digits, can never be used as a number. If you attempt to perform a mathematical operation using a string or string variable, you will get an error message. (Lesson 30 discusses BASIC statements that can be used to convert a string of digits to a number, and vice versa.)

As you become familiar with BASIC, you will regularly encounter the use of the dollar sign. The dollar sign always refers to a string or string operation and is "pronounced" string. For example, the variable *ITEM\$* is pronounced "item string," not "item dollar."

One of the more interesting operations that can be performed with strings is called *concatenation*. Concatenation is simply a large word for the process of joining strings together. The addition (+) sign is used to instruct the computer to concatenate strings.

```

A$=TYPE$+" "+ITEM$
Ok
PRINT A$
Color Television
Ok

```

Concatenation always adds strings in left-to-right order.

```

Z$=ITEM$+" ", "+TYPE$
Ok
PRINT Z$
Television, Color
Ok

```

A sequence of strings joined by the concatenation operator is referred to as a *string expression*.

Technical Note: The computer automatically reserves 256 bytes of memory for storing string data when BASIC is selected initially. If you exceed this limit, the computer will display an ?OS Error (Out of String space) message. For details on reserving more memory for string data, refer to Lesson 31.

SUMMARY

Textual information can be manipulated by BASIC if it is entered in the form of *strings*. Strings must be enclosed in quotation marks and can consist of any collection of alphabetic letters, signs, symbols, or digits from 0 to 255 characters in length.

Strings can be either *string constants* or *string variables*. String variables must conform to the same rules as numeric variables, with the added restriction that the last character of the variable name must be a dollar sign.

A string of zero length is a special string referred to as a *null* string.

Strings can be joined together using the addition (+) sign. This operation, referred to as *concatenation*, always combines strings in left-to-right order. A sequence of strings joined by the concatenation symbol is referred to as a *string expression*.

EDITing Programs

Your computer has a number of features designed to make the editing of programs simple and convenient. If you have read the owner's manual supplied with the computer, you are probably familiar with these features and have been using them all along. If not, this lesson provides a quick overview of the friendly editing features of the computer.

In Lesson 3, when you typed the "grade" program, you were instructed to check that each line was typed correctly before you pressed **ENTER**. If you observed a typing error after pressing the **ENTER** key, you were told to retype the line correctly. Because the retyped line and the erroneous line had the same line number, the corrected line replaced the incorrect line.

You do not have to retype a program line every time you want to change it. The **EDIT** command permits you to change only those

parts of a line that you wish to alter—no unnecessary retyping is needed.

Clear the BASIC workspace by typing **NEW** and pressing **ENTER**. Then enter the following program, mistakes and all.

```
100 NEWT="small salamander"
110 PRINT "A newt is a ";NEWT
120 PRINT "that can live on land and in water"
```

If you **RUN** this program as shown, the computer will respond with ?SN Error in 100. This message indicates that there is a *syntax* error in line 100. The computer displays a ?SN Error message when a program instruction does not follow the rules (syntax) of BASIC. Notice that the occurrence of any error causes program execution to stop.

The obvious error in line 100 is the lack of a dollar sign following the string variable NEWT. To correct this error, type **EDIT** and press **ENTER**. The result is shown below.

```
100 NEWT="small salamander"<
110 PRINT "A newt is a ";NEWT<
120 PRINT "that can live on land and in
water"<
←
```

In **EDIT** mode, the computer displays a ◀ symbol (◀ on the PC-8201A) to mark the end of a BASIC line. The ← symbol (◀ on the PC-8201A) marks the end of all the lines being **EDIT**ed.

To correct the variable name, move the cursor to the = sign following the word NEWT using the **→** key. Then type \$. The computer will insert the \$ sign following the T of NEWT. In **EDIT** mode, the computer automatically inserts whatever you type at the location marked by the cursor.

The same mistake is repeated in line 110, so move the cursor to the end of line 110 by pressing **DOWN** and then **CTRL** **→** (press the **CTRL** **→** combination by pressing and holding **CTRL** and then pressing **→**). Then type \$.

The **CTRL** **→** combination instructs the computer to direct the cursor to the end of the line. (The **CTRL** **←** combination moves the cursor to the start of a line.)

Now that you have made the intended corrections, press **F8** on the Models 200 and 100, or **F-10** (**SHIFT** **F-5**) on the PC-8201A to leave the **EDIT** mode. The message *Wait* is briefly flashed at the bottom left of the display while the computer stores the edited program.

RUN the program again.

```
75N Error in 100
```

There is still a syntax error in line 100. It is the use of the word **NEWT\$** as a variable. This is an invalid variable name because it contains the reserved word **NEW**. Apply the following steps to correct this error:

1. Type **EDIT** and press **ENTER**.
2. Move the cursor to the E of the first **NEWT\$**. Press the **SHIFT** **DEL** combination to delete the E.
3. Move the cursor to the E of the second **NEWT\$**. Press the **SHIFT** **DEL** combination to delete the E.
4. Press **F8** or **F-10** to leave the **EDIT** mode.

If you now **LIST** the program, it should appear as shown below.

```
100 NWT$="small salamander"
110 PRINT "A newt is a ";NWT$
120 PRINT "that can live on land and in
water"
```

The program should **RUN** without errors.

```
RUN
A newt is a small salamander
that can live on land and in water
Ok
```

In the preceding example, you edited the entire program by typing **EDIT**. You can edit selected line numbers by following the **EDIT**

command with a *line-number-range* parameter. As with the **LIST** command, the *line-number-range* parameter can be either a single number or a range of line numbers specified by a beginning line number, a hyphen, and an ending line number. A table of valid **EDIT** commands is shown below.

Command	Meaning
EDIT	Edit the entire program
EDIT 110	Edit line 110
EDIT 100-500	Edit lines 100 through 500, inclusive
EDIT -500	Edit all lines up to and including line 500
EDIT 120-	Edit all lines from 120 on, including line 120

Once the computer is placed in the **EDIT** mode, the commands available in the built-in TEXT program are available for editing the program. These include the search, copy, and paste functions. Refer to the manual supplied with your computer for a fuller discussion of the TEXT editing key sequences.

SUMMARY

The **EDIT** command permits you to change program lines without retyping the lines. The command can be used singularly or with a *line-number-range* parameter. When **EDIT** is used by itself, the entire program is available for editing. When used with a *line-number-range* parameter, the editable lines begin and end with a specified line number.

The same editing commands available in the TEXT program are available for editing BASIC programs by using the **EDIT** command. A summary of the cursor control commands is given below.

Key Sequence	Function
	Moves cursor one character to the right
 	Moves cursor to the first character of the next word

CTRL **→**

Moves cursor to the end of the current line

←
SHIFT **←**

Moves cursor one character to the left
Moves cursor to the first character of the preceding word

CTRL **←**

Moves cursor to the beginning of the current line

↑
SHIFT **↑**
CTRL **↑**

Moves cursor up one line
Moves the cursor to the top of the display
Moves the cursor to the beginning of the edited material

↓
SHIFT **↓**

Moves the cursor down one line
Moves the cursor to the bottom of the display

CTRL **↓**

Moves the cursor to the end of the edited material

To leave the **EDIT** mode, press **F8** on the Models 200 and 100, or **F-10** (**SHIFT** **F-5**) on the PC-8201A.

REVIEW TEST 1

1. What is the effect of **PRINT** 1,,3? Is this a legal sequence?
2. What is the function of the comma separator in an *expression list*? The semicolon?
3. Which of the following do you think are valid **PRINT** statements?
 - (a) PRINT ,X
 - (b) PRINT "tes";;;;;"t"
 - (c) PRINT ,,,,,,7
 - (d) PRINT "1,3"
 - (e) ? PRINT
4. What is the initial value of a numeric variable? Of a string variable?
5. Which of the following are valid assignment statements?

(a) A\$=" (b) 2 T\$="," (c) 100meters= FEET*0.3048	(d) 8=4+4 (e) let G\$= "exo"+"gust"
--	---

- | | |
|--------------------|-------------------|
| (f) 12 F=F-F | (i) MOVE\$="P-K4" |
| (g) PAY=WAGES*RATE | (j) 50 D\$=OLD\$+ |
| (h) 2=t | NEW\$ |

6. Which of the following are valid variable names?

- | | |
|--------------|-----------|
| (a) DOLLAR\$ | (g) @\$ |
| (b) ACREINCH | (h) PHd\$ |
| (c) AVE. | (i) C-10 |
| (d) start | (j) TREND |
| (e) Z1Z1Z | (k) ENTER |
| (f) BASIC | (l) 1st |

7. What are the rules for creating variable names?

8. Can 0 be a line number?

9. How can you insert a new program line in a program? Is it possible to write a program in which a new line cannot be inserted?

10. How do you replace a line?

11. How do you erase a program? A line?

12. What is the function of the **EDIT** command?

L E S S O N

6

INPUTing Information

The word "input" is used almost constantly by computer programmers and refers to the entry of information into a computer *during the execution of a program*. Naturally enough, the BASIC statement for this operation is **INPUT**.

Clear the BASIC workspace by entering **NEW** and then type in the following illustrative program.

```
100 INPUT A
110 PRINT "A=";A
```

RUN the program.

```
RUN
?
```

The question mark that you see is generated by the **INPUT** statement and is an example of a *prompt*. A prompt is a sign or message

displayed by the computer (or program) indicating that it is waiting for information. Respond to this prompt by typing 3313 and pressing **ENTER**.

```
? 3313
A= 3313
Ok
```

The program accepts what you type and assigns it to the variable that follows the **INPUT** statement. (The variable after **INPUT** is not optional.) The second line displays the value of the variable. The **INPUT** statement, then, is an assignment statement that allows you to enter the value of a variable during program execution.

As an experiment, **RUN** the program again and enter the word "test" in response to the prompt.

```
RUN
? test
Redo from start
?
```

The Redo from start message indicates that you did not enter the kind of information that can be assigned to a numeric variable. The error has not stopped program execution, however. The computer redisplay the ? prompt so that you can enter the correct kind of value. In fact, until you enter a number or press the **CTRL** **C** key combination, the computer will refuse to proceed beyond the **INPUT** statement.

Technical Note: Program execution can also be halted by pressing **SHIFT** **BREAK** on the TRS-80 Models 200 and 100, or **STOP** on the NEC PC-8201A.

If you want to input string information, use a string variable following the **INPUT** statement:

```
100 INPUT A$
110 PRINT "A$=";A$
```

(These lines will replace previous lines 100 and 110.)

You can now enter string information without creating an error condition.

```
RUN
? qwerty
A$=qwerty
Ok
```

A prompt such as a question mark is too general for most applications. Although it does indicate that you are expected to enter something, it gives no hint as to what information to enter. As a rule, it is better to provide prompts that describe what the program user is expected to input.

To facilitate the display of descriptive prompts, the **INPUT** statement has a built-in display capability. You can use this feature by placing the prompt that you want to display in quotation marks and locating it after the **INPUT** statement and before the input variable. A semicolon must separate the prompt from the variable.

```
NEW
100 INPUT "Enter a number";A
110 PRINT "The number is ";A
120 INPUT "Enter a string";A$
130 PRINT "The string is ";A$
```

After **LISTing** the program, be sure you have entered it correctly by **RUNning** it.

```
RUN
Enter a number? 186284
The number is 186284
Enter a string? first violin
The string is first violin
Ok
```

RUN the program several times. The **INPUT** statement will be used in almost every program you write, so you will want to be certain that you understand its operation.


```

RUN
Enter a number? -35.01
The number is -35.01
Enter a string? Joe Green
The string is Joe Green
Ok
RUN
Enter a number? 111.0999
The number is 111.0999
Enter a string? A PENNY SAVED
The string is A PENNY SAVED
Ok

```

The **INPUT** statement has another useful feature: a single **INPUT** statement can be used to accept multiple inputs. To use this feature, follow the **INPUT** statement with additional variables, separating the additional input variables with a comma. The following program uses this feature of the **INPUT** statement to compute the volume of a box.

```

NEW
100 INPUT "Length, Width, Height";L,W,H
110 V=L*W*H
120 PRINT "Volume =";V;"cubic feet"

```

RUN the program.

```

RUN
Length, Width, Height? 97,17,9.5
Volume = 15665.5 cubic feet
Ok

```

In the example shown above, the values for length, width, and height were entered at the same time, separated by commas. You can also enter them one value at a time, as shown below.

```

RUN
Length, Width, Height? 33
?? 13

```

```
?? 30
Volume = 12870 cubic feet
Ok
```

Observe that the computer displays two question marks for the second and third prompts to indicate that you are responding to a multiple input.

Technical Note: The **INPUT** statement is also used to read information from a file or peripheral device. Refer to Lessons 38 and 39 for details on this use of the statement.

SUMMARY

The purpose of the **INPUT** statement is to allow the entry of numeric or string information during program execution. The information entered is assigned to the variable following the **INPUT** statement. The type of input variable used (numeric or string) determines the type of information accepted by the **INPUT** statement.

If used without any of the optional parameters, the **INPUT** statement displays a question mark *prompt* when executed. A prompt is a sign or message indicating that the computer is waiting for keyboard information. To permit a program to display more informative prompts, the **INPUT** statement has a built-in display capability. The general form of the **INPUT** statement when used with the prompt option is

```
INPUT prompt;variable
```

A semicolon must separate the *prompt* from the *variable*.

The **INPUT** statement can accept multiple inputs. The general form of the **INPUT** statement when used with multiple inputs is

```
INPUT prompt;variable,variable,variable . . .
```

A comma must separate the additional variables.

Program Branching—The GOTO Statement

The normal sequence of program execution is from the smallest to the largest line number. This order is easily altered, however, by the use of the **GOTO** statement. The **GOTO** statement has the general form

GOTO *line number*

and instructs the computer to transfer program execution from the current program line to the line number, or *transfer address*, following the **GOTO** statement. For example, executing

50 **GOTO** 10000

transfers program execution from line 50 to line 10000, skipping all program lines in between. Of course, a line number matching the transfer address must exist elsewhere in the program or an error condition will occur.

The advantage of the **GOTO** statement is that it allows a program to control the order in which program lines are executed. If necessary, a program can skip the execution of a line or group of lines, or, as in the next example, repeat a sequence of lines.

Enter and **RUN** the following program.

```
NEW
100 X=0
110 PRINT X
120 X=X+1
130 GOTO 110
RUN
```

The computer begins madly counting by ones. This program instructs the computer to display the value of X, add one to X, and then go to line 110, which repeats the sequence. A program segment that is executed over and over such as this is known as a *loop*.

In this example, the computer has been placed in an *infinite loop*. It will execute lines 110, 120, and 130 until either its batteries discharge or . . . ?

Fortunately, computer manufacturers are aware of the ease with which computers can be placed in infinite loops and have provided a quick method of manually halting or “breaking” a program—the **CTRL** **C** key combination. The **CTRL** **C** key sequence instructs the computer to stop what it is doing and return to command mode.

As an alternative to **CTRL** **C**, you can press **SHIFT** **BREAK** on the Models 200 and 100, or **STOP** on the PC-8201A.

To satisfy yourself that you can always stop a running program, **RUN** and break this program several times. You will notice that whenever you press **CTRL** **C** the computer displays Break in XXX, where XXX represents the line number in which the break occurred.

Here is an interesting modification: Break the program and change line 110 so that it has a semicolon following the variable X, as shown below.

```
110 PRINT X;
```

(The program as it should now look is listed below. **LIST** your program to ensure that you have made no entry mistakes.)

```
LIST
100 X=0
110 PRINT X;
120 X=X+1
130 GOTO 110
```

RUN the program. Now, instead of displaying each value on a new line, the computer displays successive values horizontally.

As another experiment, change the semicolon in line 110 to a comma, as shown below.

```
110 PRINT X,
```

When you now **RUN** the program, the computer displays the values of X in the two print zones located on each line, rather than adjacently as specified by the semicolon.

Whenever you place a semicolon or a comma after a **PRINT** statement, you create a *pending print state*. A pending print state causes the computer not to advance to a new line of the display when the next **PRINT** statement is executed (unless the line is full).

Counting examples serve primarily to demonstrate visually the looping power of the **GOTO** statement. A more practical application of a loop is shown below.

```
NEW
100 INPUT "FEET";FEET
110 METERS=FEET*.3048
120 PRINT "Meters =";METERS
130 GOTO 100
```

This program performs a simple function. It takes the value you enter in response to "Feet?" and converts it to meters. By performing the function in a loop, however, you avoid having to reenter

RUN to convert additional values. The program automatically loops back to the **INPUT** prompt until you press **CTRL C** to stop program execution.

```
RUN
Feet? 44
Meters = 13.4112
Feet? 99.9
Meters = 30.44952
```

(Press **CTRL C** to stop program execution.)

This technique is often used when an application requires repetitive work.

SUMMARY

The **GOTO** statement instructs the computer to transfer program execution from the current line to the line number following the **GOTO** statement.

The **GOTO** statement can be used to skip the execution of a line or group of lines, or to place a program in a *loop*. Most programs contain at least one loop, although, as a rule, not an *infinite loop* as in the examples in this lesson. An infinite loop is a sequence of program instructions that are executed over and over without any provision for exit. The decision-making capabilities of the computer discussed in the next lesson make it possible to write programs that control the number of times that a loop is executed.

An infinite loop can always be stopped manually by pressing **CTRL C**. The **CTRL C** key combination will stop any running program, not just a program in an infinite loop. (As an alternative to **CTRL C**, you can press **SHIFT BREAK** on the TRS-80 Models 200 and 100, or **STOP** on the NEC PC-8201A.) Placing a semicolon or a comma at the end of a **PRINT** statement instructs the computer not to advance to the next line of the display when the next **PRINT** statement is executed. This is referred to a *pending print state*.

Decision Making— The IF THEN Statement

The program statements discussed so far have all had one characteristic in common—they are always executed when encountered in a program. With the **IF THEN** statement, however, the execution of a statement, line, or group of lines can be based on the result of a decision-making test.

The general form of the **IF THEN** statement is

IF *condition* THEN *action*

where *condition* consists of a decision-making test and *action* is any valid program statement.

The decision-making tests discussed in this lesson are the *relational* tests. Lesson 33 discusses the use of *logical* decision-making tests.

Six relational tests are available. The symbols used to perform them in BASIC and their meaning are listed below. Notice that the not-

equal-to, less-than-or-equal-to, and greater-than-or-equal-to symbols are made by combining two other symbols. (The more usual symbols for these operations, $\neq$, $\leq$, and $\geq$, respectively, do not exist on most computer keyboards.)

Symbol	Meaning
<	less than
>	greater than
=	equal to
<>	not equal to
<=	less than or equal to
>=	greater than or equal to

A relational test can have only two possible results: the relation is either true or it is false. If the relation is true, the action is performed. If the relation is false, the action is skipped. This rule can be summarized as "do if true, skip if false."

The following program illustrates how the **IF THEN** statement is combined with a relational test to make program decisions. The program begins by requesting that you enter a number. When you do, the **IF THEN** statement compares the number with zero. If your entry is less than zero (relation true), the **PRINT** statement is executed. If your entry is not less than zero (relation false), the **PRINT** statement is skipped.

```
NEW
100 INPUT "Enter a number";N
110 IF N<0 THEN PRINT "The number is negative!"
120 GOTO 100
```

RUN the program and try various numeric entries. Whenever a value less than zero is entered, the program displays "The number is negative!"

```
RUN
Enter a number? -7
The number is negative!
```



```

Enter a number? 7
Enter a number? -.000001
The number is negative!
Enter a number?

```

(Press **CTRL** **C** to stop program execution.)

As it stands, the program only informs you when a number is negative. With several additional decision-making tests, you can make the program tell you whether your entry is negative, positive, or zero.

```

112 IF N>0 THEN PRINT "The number is positive!"
114 IF N=0 THEN PRINT "The number is zero!"

```

RUN the expanded version of the program.

```

RUN
Enter a number? 99
The number is positive!
Enter a number? -.99
The number is negative!
Enter a number? 0
The number is zero!

```

(Press **CTRL** **C** to stop program execution.)

This program analyzes the number that you enter by “filtering” it through three decision-making tests. Since the three tests are mutually exclusive, only the **PRINT** statement following the true test is executed. The other **PRINT** statements are skipped.

Technical Note: It is the action that follows the **THEN** part of the **IF THEN** statement that is skipped if the relation is false, not the relational test. The relational test is always performed.

The next program illustrates how an **IF THEN** statement can be used to control the number of times a loop is executed. The program calculates the average of a series of numbers that you input.

```
NEW
100 SUM=0
110 INPUT "Number of entries";N
120 COUNT=N
130 INPUT "Entry";ENTRY
140 SUM=SUM+ENTRY
150 COUNT=COUNT-1
160 IF COUNT=0 THEN GOTO 180
170 GOTO 130
180 AVERAGE=SUM/N
190 PRINT "Average =";AVERAGE
```

The program begins by asking how many numbers you intend to enter and assigning that value to the variable COUNT. Next, the program requests your entry. The value that you enter is added to the variable SUM (which was set to 0 at the start of the program). Then, after subtracting one from COUNT, the program tests if COUNT equals zero. If COUNT does equal zero (relation true), execution is transferred to line 180 to calculate and display the average. If COUNT does not equal zero (relation false), the conditional action is skipped, allowing execution to advance to line 170, which sends the program back to line 130 to get another entry.

A sample **RUN** of the program is shown below.

```
RUN
Number of entries? 5
Entry? 10
Entry? 20
Entry? 30
Entry? 40
Entry? 50
Average = 30
Ok
```

This program illustrates a common programming technique. The computer is placed in a loop until a counting variable reaches a desired terminating value; then program execution is transferred out of the loop. The purpose of the counting variable is to control

the number of times the loop is executed. In this example, the counting variable is set initially to the number of entries and is decreased by one each time a new entry is accepted (a "countdown"). As later examples illustrate, it is also possible to count up to a desired value.

An interesting point to observe is that the decision used to end the loop can be written in two ways. In this example, the program tests whether the value of COUNT equals zero. By modifying the program to test whether the value of COUNT is *not* equal to zero, the program can be made shorter and more efficient.

```

100 SUM=0
110 INPUT "Number of entries";N
120 COUNT=N
130 INPUT "Entry";ENTRY
140 SUM=SUM+ENTRY
150 COUNT=COUNT-1
160 IF COUNT<>0 THEN GOTO 130
170 AVERAGE=SUM/N
180 PRINT "Average =";AVERAGE

```

This program is functionally equivalent to the preceding program. Instead of using the **IF THEN** statement to transfer program execution out of the loop, this version of the program uses the **IF THEN** statement to keep the program looping until the exit condition is met.

In both these examples, the action following the **IF THEN** statement was a **GOTO** statement. One of the features of the **IF THEN** statement is that when a **GOTO** statement is used as a conditional action, the **GOTO** can be dropped and only the line number need be supplied. Thus, a statement such as

```
160 IF COUNT<>0 THEN GOTO 130
```

can also be entered as

```
160 IF COUNT<>0 THEN 130
```

BASIC recognizes the implied **GOTO** between the **THEN** and 130.

SUMMARY

The **IF THEN** statement provides a BASIC program with the capability to perform different actions depending on the result of a decision-making test. If the test is true, the action following the **THEN** portion of the **IF THEN** statement is executed. If the test is false, the action is skipped.

The decision-making tests discussed in this lesson are listed below.

Symbol	Meaning
<	less than
>	greater than
=	equal to
<>	not equal to
<=	less than or equal to
>=	greater than or equal to

A common application of decision making is to control the number of times that a loop is executed. The usual technique is to maintain a counting variable that is either increased or decreased by one each time the loop is executed until a desired terminating value is reached. When the counting value reaches the terminating value, execution is transferred out of the loop with a **GOTO** statement.

When a **GOTO** statement is used as the action parameter of an **IF THEN** statement, the **GOTO** can be dropped and only the line number entered. For example, both

```
100 IF A=10 THEN 150
```

and

```
300 IF A<>120 THEN 90
```

are valid **IF THEN** statements that use this option. The computer understands the implied **GOTO** in these lines.

Defining the BASIC Function Keys

One of the major features of your computer is the abbreviated entry of BASIC instructions permitted by the function keys. The function keys are labeled **F1** through **F8** on the TRS-80 Models 200 and 100, and **F-1** through **F-5** on the NEC PC-8201A. (In spite of appearances, the PC-8201A has two more function keys than the Models 200 or 100. Functions F-6 through F-10 are accessed on the PC-8201A by pressing the **SHIFT** key in conjunction with the **F-1** through **F-5** keys. For example, F-7 is obtained by pressing **SHIFT F-2**.)

The computer permits you to assign whatever BASIC instructions you want, up to 15 characters in length, to any function key. You can then type in that instruction by simply pressing the function key.

Technical Note: Each of the computer's built-in programs has different uses for the function keys. This lesson discusses the function

keys as used in BASIC. Changing the definitions of the function keys in BASIC does not alter the definitions of the keys when used in the TEXT, TELECOM, and ADDRSS programs.

When you first start up your computer, BASIC assigns certain predefined or "default" instructions to the function keys. On the Models 200 and 100, these instructions, or any new ones you have defined, can be displayed by executing the **KEY LIST** command. The key definitions are displayed in a two-column format, as shown below. If a function key does not have an assigned definition, the corresponding location in the list is left blank.

Function key 1 Function key 2

Function key 3 Function key 4

Function key 5 Function key 6

Function key 7 Function key 8

A shortened form of these definitions can be displayed on the bottom line of the screen by executing the **SCREEN** command. On the Models 200 and 100, the first four characters of all eight instructions are displayed. On the PC-8201A, the first six characters of the definitions of function keys F-1 through F-5 are displayed. To see the definitions of functions F-6 through F-10, you must press the **SHIFT** key.

The **SCREEN** command has the general form

SCREEN 0, *on/off*

The *on/off* parameter determines whether the definitions are displayed or erased. If the *on/off* value is 0, the definitions are erased. If the *on/off* value is 1, the function key definitions are displayed.

On the Models 200 and 100, the function key definitions can also be turned on or off by pressing the **LABEL** key.

Although the designers of the computer have made good selections for the default definitions of these keys, you will find that their choices do not always meet your needs. Because of the great convenience of entering multiple-character instructions with a single keystroke, you will find it advantageous to assign to a function key any instruction that you intend to type often.

You assign or change the definition of a function key by executing the **KEY** command. The **KEY** command has the general form

KEY *key number, string expression*

where *key number* is the number of the function key that you want to define and *string expression* is the definition to assign to the key.

To illustrate the use of this command, assign the **INPUT** statement to function 7 by typing

```
KEY 7, "INPUT"
```

and pressing **ENTER**. Since **INPUT** is a string constant, it is enclosed in quotes.

Now, when you want to type **INPUT**, simply press **F7** (**SHIFT F-2** on the PC-8201A).

To replace the definition of this key with another, for example with the **GOTO** statement, enter

```
KEY 7, "GOTO"
```

If you want to erase (rather than replace) the definition of a function key, assign the null string to the key, as illustrated below.

```
KEY 7, ""
```

Technical Note: You can add quote (") marks and **ENTER** instructions to function definitions using the **CHR\$** function. For details regarding the use of this instruction, refer to Lesson 29.

SUMMARY

The computer permits you to assign whatever BASIC instructions you want, up to 15 characters in length, to any function key. You can then type in that instruction by simply pressing the function key.

The **KEY** command is used to assign or change the definition of a function key. The **KEY** command has the general form

KEY *key number,string expression*

where *key number* is the number of the function key that you want to define and *string expression* is the definition to assign to the key.

A shortened form of the function key definitions can be displayed on the bottom line of the screen by executing the **SCREEN** command. The **SCREEN** command has the general form

SCREEN 0,*on/off*

The *on/off* parameter determines whether the definitions are displayed or erased. If the *on/off* value is 0, the definitions are erased. If the *on/off* value is 1, the function key definitions are displayed.

On the Models 200 and 100, the function key definitions can also be turned on and off by pressing the **LABEL** key.

L E S S O N 10

Using RAM Files

As you probably have observed by now, each programming example in this book begins with **NEW** to clear the BASIC workspace memory. This ensures that you do not inadvertently mix program lines from an old program with the new example, but does not enable you to retain the old program. If you want to preserve a program for later study or use, you should **SAVE** it before you clear the BASIC workspace.

The **SAVE** command instructs the computer to store a program in memory as a separate unit, or *file*. Once you have saved a program, you can clear the BASIC workspace without erasing the saved program.

Clear the BASIC workspace and enter the following sample program:

```
NEW
Ok
100 PRINT "TEST"
```

Next, **SAVE** the program by entering

```
SAVE "SAMPLE"
Ok
```

Now clear the BASIC workspace. Verify that the program is no longer in the workspace by listing the program.

```
NEW
Ok
LIST
Ok
```

You can confirm that the file has been saved by executing the **FILES** command. The **FILES** command instructs the computer to list the names of the files in memory, including any files that you have created with the TEXT program. (Only RAM files are listed; the built-in ROM files are not displayed.)

```
FILES
INTRO .DO   CHAP1 .DO   CHAP2 .DO
CHAP3 .DO   TEST1 .DO   CHAP7 .DO
CHAP10.D0   SAMPLE.BA
Ok
```

Notice the last file on the list: SAMPLE.BA. This is the SAMPLE program file. (Except for SAMPLE.BA, the files on your list will differ from those shown in the example. Your screen will show the files stored in your machine.)

The .BA that is added to the file's name is known as an *extension*. A file extension provides a simple method of grouping files into categories. A .BA extension identifies a file as a BASIC file. [A .DO extension identifies a file as a TEXT (document) file.] If you do not specify it yourself, the computer automatically adds a .BA extension to the name you enter when you **SAVE** a program.

Technical Note: The **SAVE** command can also transfer a copy of a program to an external device such as a cassette recorder, printer, or another computer. The discussion in this lesson is limited

to saving programs in RAM. For details on saving programs on cassette tape, refer to Lesson 25.

When you are ready to use a "saved" program again, you can either:

1. Select it from the Menu Screen by moving the cursor to the file's name and pressing **ENTER**.
2. Use the **LOAD** command to "load" the program into the BASIC workspace.

To illustrate the **LOAD** command, load the program back into the workspace and then **LIST** it to verify its presence.

```
LOAD "SAMPLE"
Ok
LIST
100 PRINT "TEST"
Ok
```

Technical Note: The **LOAD** command can "load" a program from an external device such as a cassette recorder or another computer. The discussion in this lesson is limited to loading programs saved in RAM. For details on loading programs from cassette tape, refer to Lesson 25.

The Model 100 can store up to 19 files in its memory. The Model 200 can store up to 46 files in each bank of its memory. The PC-8201A can store up to 21 files in each bank of its memory. Included in this limit are all .DO files that you may create. If you attempt to exceed this limit, a ?FL Error (File Limit) occurs.

Once a program has been saved, the computer maintains the program's file status even if you increase or decrease the size of the program. If you attempt to **SAVE** the program a second time, even under a new name, a ?FC Error (illegal Function Call) condition occurs.

If you want to change the name of a file, you must use the **NAME** command. The general format for the **NAME** command is

```
NAME "old filename" AS "new filename"
```

The **NAME** command requires that you specify the file extension for both the original name and the new name.

Change the name of the SAMPLE file to OMEGA. Then verify that the file's name has been changed using the **FILES** command.

```
NAME "SAMPLE.BA" AS "OMEGA.BA"
Ok
FILES
INTRO .DO    CHAP1 .DO    CHAP2 .DO
CHAP3 .DO    TEST1 .DO    CHAP7 .DO
CHAP10.D0    OMEGA .BA*
Ok
```

The asterisk following the OMEGA.BA filename indicates that OMEGA.BA is the resident (or current) file in the BASIC workspace.

When you are ready to erase a file from memory, use the **KILL** command. The **KILL** command will delete any RAM file, including .DO files. Before you can delete OMEGA.BA, however, you must remove it as the resident program. *You cannot **KILL** a file that is resident in the BASIC workspace.*

```
NEW      (To remove OMEGA.BA as the resident program)
Ok
KILL "OMEGA.BA"
Ok
FILES
INTRO .DO    CHAP1 .DO    CHAP2 .DO
CHAP3 .DO    TEST1 .DO    CHAP7 .DO
CHAP10.D0
Ok
```

The **KILL** command requires that you specify the file extension together with the filename.

Think twice before you delete a file, because *once a program is deleted, it cannot be recovered.*

SUMMARY

The **SAVE** command is used to assign a program a name and store it as a unit in memory. A "saved" program is referred to as a *file*. After a program has been saved, you can clear (**NEW**) the BASIC workspace in preparation for entering a new program without erasing the saved program.

When you are ready to use a "saved" program again, you can either:

1. Select it from the Menu Screen by moving the cursor to the filename and pressing **ENTER**.
2. Use the **LOAD** command to "load" the program into the BASIC workspace.

The **NAME** command enables you to change the name of a memory file.

The **KILL** command deletes a file from memory. Once a program is deleted it cannot be recovered.

The **FILES** command lists the names of the files in memory. If a file is resident in the BASIC workspace, the **FILES** command marks that file's name with an asterisk.

Filenames can be up to six characters in length, followed by a period and a two-character *extension*. A .BA extension identifies a file as a BASIC file. A .DO extension identifies a file as a TEXT (document) file. If you do not specify an extension when you **SAVE** a program, the computer adds a .BA extension to the filename. (The PC-8201A permits you to intermix upper and lowercase letters in filenames. The Models 200 and 100 convert all lowercase letters in a filename to uppercase letters.)

The **NAME** and **KILL** commands require that you specify the extension with the filename when you execute the command.

REVIEW TEST 2

1. Which of the following are valid **INPUT** statements?
 - (a) 5 INPUT ENTER YOUR NAME;n\$
 - (b) 300 INPUT a;b;c;d;e
 - (c) 110 INPUT "Account"; ACCOUNT+2
 - (d) 1 INPUT ID\$+"-001";ITEM\$
 - (e) 50 INPUT "Enter five names"; N1\$,N2\$,N3\$,N4\$,N5\$
 - (f) 22 INPUT "Direction", D\$
2. Write a program that counts by 9's and displays the count in the first and fifteenth character positions (zones). Modify the program to count backward by 99 beginning at 2475, displaying the values adjacently.
3. What is a *pending display state*? How is it created, and why is it useful?
4. The first two lines of a program designed to balance a checking account are given below. Assume that deposits are entered as positive values and checks as negative values. Complete this program so that you can use it to balance your checkbook.

```

100 INPUT "Current balance";BALANCE
110 INPUT "Amount of check or deposit";AMOUNT

```

5. What is a "counting variable," and how is it used to control the number of times a loop is executed?
6. Which of the following are valid **IF THEN** statements?
 - (a) 100 IF X=X+1 THEN 300
 - (b) 121 IF TRUE=FALSE THEN FALSE=TRUE+2
 - (c) 112 IF T><12 THEN INPUT"ENTER A NEW
NUMBER";T
 - (d) 50 IF X<Y<Z then 200
 - (e) 75 IF 2*B-C/12<=2400/C THEN ID\$="end"
 - (f) 92 IF COUNT + 1 = 99 THEN GOTO 2001
 - (g) 100 IF YARDS>=35 THEN IF DOWN=4 THEN PRINT
"BETTER KICK!"
7. A wholesale distributor of chess sets offers the following discount schedule: 20 percent off on orders of 100 or fewer sets, 25 percent off on orders between 101 and 500 sets, 30 percent off on orders between 501 and 1000 sets, 35 percent off on orders between 1001 and 4999 sets, and 40 percent off on orders of 5000 or more sets. The base cost of the chess sets is \$14.88 per set. Write a program that allows you to input the number of chess sets ordered and displays the total cost of the order. Use the program to calculate the wholesale cost of 433 sets, 888 sets, and 2001 sets.
8. What are file *extensions*, and why are they useful? Which file manipulation commands require that you specify the extension with the filename?
9. What is the purpose of the following BASIC commands?

- (a) KEY 1, "FILES"
- (b) NAME "CARDS.BA" AS "DRAW.BA"
- (c) KILL "MENU.BA"
- (d) SAVE F\$
- (e) LOAD "ASSETS"
- (f) FILES
- (g) RUN "SOUND.BA"

Order of Calculations

A fundamental rule of mathematics is that a series of mathematical operations can have only one result. Without such a convention, problems that intermix different mathematical operations could have several answers, depending on the order in which the operations are performed. Consider, for example, the problem

$$3 + 5 \times 2 + 4 =$$

If the addition operations are performed before the multiplication operation, the result is $8 \times 6 = 48$. If the multiplication operation is performed before the addition operations, the result is $3 + 10 + 4 = 17$.

The correct answer to this problem according to the rules of mathematics is 17, the answer given by the computer if you **PRINT** the value of the expression just as it is written. To obtain this answer, the computer must perform multiplication before addition. This capa-

bility to perform one operation before another is achieved by a built-in ranking system and a set of rules governing the order in which operations are executed. These rules are usually referred to as *levels of precedence*.

The levels of precedence used by the computer in order of highest to lowest priority are as follows:

Precedence	Operation Name	Operator Symbol
1	Higher-order functions	(see discussion below)
2	Exponentiation	$\wedge$
3	Unary minus (negation)	$-$
4	Multiplication and division	$*$ and $/$
5	Integer division	$\backslash$
6	Modulus arithmetic	MOD
7	Addition and subtraction	$+$ and $-$

The higher-order functions assigned to the first precedence level consist of mathematical operations such as natural logarithms, trigonometric functions, and absolute value, as well as operations unique to BASIC such as **INT**, **RND**, and **LEN**. Because of the variety and complexity of these functions, they are discussed in later lessons (see Lessons 13, 28, 29, and 30). For the moment, be aware that when intermixed with other operations from this table, higher-order functions are performed first.

The second priority level is assigned to exponentiation. Exponentiation means "raised to a power" and refers to operations of the general form

base number^{exponent}

A familiar example of exponentiation is $5^2 = 25$.

The exponentiation operator used by the computer is the ^ symbol, obtained by pressing **SHIFT** 6.

```
PRINT 9^.5
3
Ok
PRINT 121^4
214358881
Ok
```

The third precedence level is assigned to the unary operation "taking the negative of" as in the example

```
X=9
PRINT -X
-9
Ok
```

This type of minus operation is called *unary*, meaning "one," to emphasize its difference from the operation of subtraction, which operates on two values.

The fourth precedence level is assigned to multiplication and division.

The fifth precedence level is assigned to an operation known as *integer division*. Integer division is "normal" division with these restrictions:

1. The dividend and divisor must be in the range -32768 to 32767. If you attempt to perform integer division with a value outside this range, an **OV Error** (overflow) occurs. If the dividend or the divisor falls within this range but is not a whole number, BASIC rounds the value to a whole number before performing the division operation.
2. The result (quotient) of the operation is always truncated to a whole number.

The symbol used for integer division is the backslash (\) character. (The backslash is obtained on the Models 200 and 100 by pressing the **GRAPH** **—** key combination.)

```

PRINT 1/7
    .14285714285714
Ok
PRINT 1\7
    0
Ok

```

Observe that the result of the integer division of 1 by 7 is zero, due to the truncation of the answer (truncation is the dropping of any decimal portion of a number).

The sixth precedence level is assigned to *modulus arithmetic*. The general form of the operation is

value1 MOD value2

Modulus arithmetic gives the whole-number value that is left over (the *remainder*) after the integer division of *value1* (the dividend) by *value2* (the divisor).

```

PRINT 7 MOD 3
    1
Ok

```

The remainder is 1.

Because the **MOD** operation uses integer division to calculate the remainder, *value1* and *value2* must be within the range -32768 to 32767.

The seventh precedence level is assigned to addition and subtraction.

When the computer evaluates a calculation (hereafter referred to as a *numeric expression*), it works through the expression from left to right performing: first, all higher-order functions in the expression; second, all exponentiations; third, all unary minuses; fourth, all multiplications and divisions; fifth, all integer divisions; sixth, all **MOD** operations; and finally, all additions and subtractions. When two or more operations of the same precedence are present, the leftmost operation is performed first.

This order of precedence is followed whether an expression is evaluated in command mode or within a program.

The computer's capability to recognize and perform mathematical operations in the order expected by mathematicians is a great convenience. Sometimes it is necessary, however, to evaluate an expression in a different order.

If you want an expression to be evaluated in another order, put parentheses around the part of the expression you want evaluated separately. By enclosing a portion of the expression in parentheses, you instruct the computer to evaluate the operations within the parentheses before any operations outside the parentheses. For example, entering

```
PRINT 2^(3+10/2)
256
Ok
```

forces the computer to perform the division and addition operations before the exponentiation operation. Observe that within the parentheses, the normal rules of precedence still apply.

A calculation can require parentheses to be evaluated correctly even though it is not written with parentheses. For example, you will get the wrong answer if you enter

$$\frac{41 + 3}{17 - 6}$$

as $41 + 3 / 17 - 6$.

In examples of this type, the numerator and denominator must be evaluated before the division operation is performed. Therefore, the correct way to enter this problem is

```
PRINT (41+3)/(17-6)
4
Ok
```

If you are not certain that the computer will evaluate an expression in the order that you want, use parentheses. If necessary, use pa-

$$2*(2*(2*(2*(2*(2+3)+2)+2)))$$

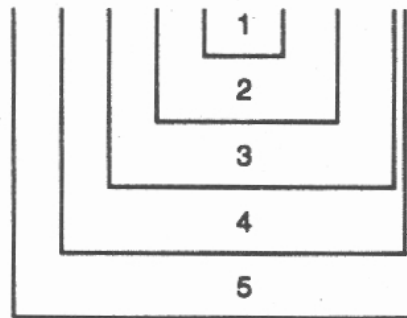


Fig. 11-1 The order of evaluation of parentheses.

rentheses within parentheses. As long as you have defined the evaluation order correctly, the presence of extra parentheses will not disturb the calculation.

When parentheses are used within parentheses, the computer begins its evaluation process with the innermost set of parentheses and works outward.

```
PRINT 2*(2*(2*(2*(2*(2+3)+2)+2)))
      208
```

The order in which this expression is evaluated is illustrated graphically in Fig. 11-1.

When using parentheses, match the pairs of parentheses carefully. If you enter an unequal number of left and right parentheses, the computer will display the error message ?SN Error.

SUMMARY

The order in which mathematical operations are evaluated is controlled by a ranking system usually referred to as the computer's *levels of precedence*. This system is based on conventions of mathematics and is designed to allow expressions that are entered as they are generally written to be evaluated in the correct algebraic order.

The levels of precedence used in evaluating mathematical operations are listed below.

Precedence	Operation Name	Operator Symbol
1	Higher-order functions	(see Lessons 12, 13, 28, and 29)
2	Exponentiation	$\wedge$
3	Unary minus (negation)	$-$
4	Multiplication and division	$*$ and $/$
5	Integer division	$\backslash$
6	Modulus arithmetic	MOD
7	Addition and subtraction	$+$ and $-$

These operations are performed in a strict left-to-right order, beginning with the highest level of precedence. If two or more operations of the same level are present in an expression, the leftmost operation is performed first.

Through the selective use of parentheses, the computer can be made to perform operations in any order. Parentheses instruct the computer to evaluate all operations within the parentheses before any operations outside the parentheses. If you have any doubts about how the computer will evaluate an expression, use parentheses.

When parentheses are used within parentheses, the computer begins its evaluation process with the innermost set of parentheses and works outward.

Higher-Order Mathematical Functions

The higher-order functions of the computer fall into two broad categories: mathematical and nonmathematical functions. This lesson briefly discusses the mathematical functions. Nonmathematical functions are discussed in Lessons 28, 29, and 30.

The mathematical functions have the general form

function(numeric expression)

where *numeric expression* represents the value operated on by the *function*. This value, which can be a constant, variable, or calculation, is referred to as the *argument* of the function. The argument of a function must always be enclosed within parentheses or an error condition will occur.

Executing a function does not change the value of its argument.

Thus,

```
PRINT SQR(X)
```

displays the square root of X without changing the value of X. Because a function gives a result without changing its argument, it is said to *return* a value.

You can use a function in almost any way that you can use a variable. You can display it, manipulate it mathematically, base decision-making tests on it, or assign its value to a variable. You can even include it in the argument of another function; for example,

```
PRINT LOG(SQR(ABS(-45)))
1.9033312448852
Ok
```

What you cannot do with a function is use it on the left side of an assignment sequence. Thus,

```
SQR(X)=11
```

is an illegal assignment sequence.

The higher-order mathematical functions are listed below.

Function	Definition
ABS(<i>numeric expression</i>)	absolute value (makes a number positive)
ATN(<i>numeric expression</i>)	arctangent
COS(<i>numeric expression</i>)	cosine
EXP(<i>numeric expression</i>)	natural antilogarithm
LOG(<i>numeric expression</i>)	natural logarithm
SIN(<i>numeric expression</i>)	sine
SQR(<i>numeric expression</i>)	square root
TAN(<i>numeric expression</i>)	tangent

As you can see from the list, the name of a function is based on what the function does.

Several examples of mathematical functions follow:

```
PRINT SIN(30)
-.98803162409273
Ok
PRINT LOG(3.14159265359)
1.1447298858494
Ok
PRINT SQR(121)
11
Ok
PRINT ABS(-9.1)
9.1
Ok
```

SUMMARY

The higher-order mathematical functions have the general form

function(numeric expression)

where *numeric expression* represents the *argument*, or value operated on, of the function. The argument can be a constant, variable, or calculation, but must be enclosed in parentheses. Because a function gives a result without altering its argument, it is said to *return* a value.

The trigonometric functions return values in radians.

The INT, FIX, and SGN Functions

The **INT**(*numeric expression*) function is a higher-order function used to find the integer portion of the value entered for *numeric expression*. The word *integer* is used here in its mathematical sense, meaning any of the natural numbers (1, 2, 3, etc.), the negatives of these numbers, or zero.

The use of the word *integer* can be confusing in BASIC, because it is used in several different senses. The mathematical sense is always used in relation to the **INT** function, as discussed in this lesson. The use of the word to mean a type of numeric data is covered in Lesson 14.

To illustrate the operation of the **INT** function, enter

```
PRINT INT(7.459)
7
OK
```

The function returns the natural (whole) number 7. For negative values, the **INT** function returns a negative integer that is *less than or equal* to the value of the argument. Thus,

```
PRINT INT(-7.459)
-8
Ok
```

displays -8 and not -7 as you might expect (-8 is less than -7.459; -7 is not).

A function that is very similar to the **INT** function is the **FIX** function. The **FIX(numeric expression)** function returns a value in which any digits to the right of the decimal point of the argument are truncated (dropped). For positive values, the **INT** and **FIX** functions return the same values.

```
X=12.3456
PRINT INT(X);FIX(X)
12 12
Ok
```

But for negative values with digits to the right of the decimal point, the functions return different values.

```
PRINT INT(-.345);FIX(-.345)
-1 0
Ok
```

The following table illustrates these differences.

Value of X	INT(X)	FIX(X)
100	100	100
1.1	1	1
-30	-30	-30
-1.8	-2	-1

The **INT** and **FIX** functions have a surprising variety of programming applications. Either function can be used, for example, to determine whether a value is even or odd.

NEW

```
100 INPUT "Sample number";N
110 IF FIX(N/2)=N/2 THEN PRINT "The number is even"
120 IF FIX(N/2)<>N/2 THEN PRINT "The number is odd"
130 GOTO 100
```

The program compares the value returned by the function after dividing the number by 2 with a straightforward division by 2. If the number is even, there is no fractional portion to discard after the division and the relation is true. If the number is odd, there is a fractional result left over and the relation is false. (Replacing **FIX** with **INT** produces the same result.)

A sample **RUN** of this program is given below.

```
RUN
Sample number? 99778
The number is even
Sample number? -333
The number is odd
```

(Press **CTRL** **C** to halt execution.)

This technique can be used any time you want to determine whether one number is evenly divisible by another. For example, if you are manipulating dates in a program, you may want to find out which years are leap years. The following program illustrates how **INT** can make this check.

NEW

```
100 INPUT "Sample year";YEAR
110 IF INT(YEAR/4)=YEAR/4 THEN PRINT "The year is a
    leap year"
120 IF INT(YEAR/4)<>YEAR/4 THEN PRINT "The year is
    not a leap year"
130 GOTO 100
```

A sample **RUN** is shown below.

```

RUN
Sample year? 1946
The year is not a leap year
Sample year? 1428
The year is a leap year

```

(Press **CTRL** **C** to stop program execution.)

The **INT** function can also be used to round values to a desired number of decimal places. The general formula for this operation is

$$R = \text{INT}(N * P + .5) / P$$

where R = the rounded value, N = the number to be rounded, and P = a power of ten such as 10, 100, 1000, or 10000. The value selected for P determines the number of places to which the value R is rounded. For example, if $P = 100$, R is rounded to two decimal places. If $P = 10000$, R is rounded to four decimal places.

Briefly, the rounding formula works as follows. The value to be rounded is first multiplied by P to shift the decimal point a fixed number of places to the right. For example, if N is 12.3456, multiplying by 100 gives 1234.56. Next, the equation adds .5 to this intermediate result. The purpose of this addition is to force decimal values of .5 to .9 to "round up" to the next higher unit. In the example quoted above, the intermediate result is now $1234.56 + .5 = 1235.06$. Next, the **INT** function is used to discard any digits to the right of the decimal point. Finally, the intermediate result is divided by P to restore the decimal point to its original position. Thus the final result in this example is $1235/100 = 12.35$, or 12.3456 rounded to two decimal places.

This formula is correct only for positive values. If a program must round negative values also, use the formula

$$R = (\text{INT}(\text{ABS}(N) * P + .5) / P) * \text{SGN}(N)$$

See the end of the lesson for a discussion of the **SGN** function.

The following program illustrates rounding numbers to two decimal places.

```
NEW
100 P=100
110 INPUT "Sample number";N
120 R=INT(N*P+.5)/P
130 PRINT "The rounded number is: ";R
140 GOTO 110
```

A sample **RUN** is shown below.

```
RUN
Sample number? 1.495
The rounded number is: 1.5
Sample number? 1.494
The rounded number is: 1.49
```

(Press **CTRL** **C** to stop program execution.)

The program correctly rounds 1.495 up and 1.494 down.

If you want the program to round to another number of decimal places, change the value of P in line 100. For example, entering

```
100 P=100000
```

will cause the program to round to five decimal places.

The **FIX** function can be used to find the fractional portion of a number. This is accomplished by subtracting the value returned by **FIX** from the number:

```
NEW
100 INPUT "Sample number";N
110 F=N-FIX(N)
120 PRINT "The fractional part is: ";F
130 GOTO 100
```

A sample **RUN** is shown below.

```

RUN
Sample number? 765.4321
The fractional part is: .4321
Sample number? -.99
The fractional part is: -.99

```

(Press **CTRL** **C** to stop program execution.)

This program correctly returns the fractional value for either positive or negative numbers.

Another higher-order function that is useful in testing the properties of a number is the **SGN**(*numeric expression*) function. The **SGN** function returns a value of -1, 0, or 1 based on the following rules:

Value of Argument	SGN Response
Less than 0	-1
Zero	0
Greater than 0	1

A program illustrating the operation of the **SGN** function is given below.

```

NEW
100 INPUT "Sample number";N
110 PRINT SGN(N)
120 GOTO 100

```

A sample **RUN** is shown below.

```

RUN
Sample number: -7
-1
Sample number? 0
0
Sample number: 7
1

```

(Press **CTRL** **C** to stop program execution.)

The **SGN** function is often used to restore the sign of a number. For example, to round both negative and positive numbers to a specific number of places, use the formula

$$R=(\text{INT}(\text{ABS}(N)*P+.5)/P)*\text{SGN}(N)$$

The rounding portion of the formula (shown in italics) is identical to that given earlier, with the addition of the absolute value (**ABS**) function to convert the number to a positive value. The **ABS** function ensures that the sign of the number does not alter the calculation.

After the rounded value has been found, it is necessary to restore the minus sign for negative numbers. One way to do this would be to use an **IF THEN** statement to test if the number was negative and then multiply by -1 when it was. It is more efficient to use the **SGN** function, however. Since **SGN** returns either -1 , 0 , or 1 , multiplying by the value of **SGN(N)** will leave the sign unaltered when N is positive (or zero), and will change it to minus when N is negative.

SUMMARY

The **INT**(*numeric expression*) function returns the integer value of *numeric expression*. For positive arguments, this is the value of the argument stripped of any fractional or decimal part. For negative arguments, this is the negative integer just less than or equal to the value of the argument.

The **FIX**(*numeric expression*) function returns a value in which any digits to the right of the decimal point of the argument are truncated (dropped).

Common applications of the **INT** and **FIX** functions include determining if one number is divisible by another, rounding numbers to a fixed number of decimal places, and finding the fractional portion of a number.

The **SGN**(*numeric expression*) function returns -1 , 0 , or 1 , depending on the sign of *numeric expression*. If *numeric expression* is

negative, **SGN** returns -1 . If *numeric expression* equals zero, **SGN** returns 0. If *numeric expression* is positive, **SGN** returns 1.

A common application of the **SGN** function is to restore the sign of a number after it has been removed by a previous calculation.

Numeric Variable Types

When you enter a statement such as

```
PRINT 10/7  
1.4285714285714  
OK
```

the computer displays the result of the calculation (a repeating decimal) to 14 significant digits. This level of accuracy is referred to in BASIC as *double precision*. Because such a high degree of precision is not required for all programming applications, BASIC provides two other levels of numeric accuracy: *single precision* and *integer precision*. Single-precision numbers are restricted to six significant digits, and integer precision numbers are restricted to whole numbers in the range -32768 to 32767, inclusive.

Note that BASIC's definition of integer precision does not include all possible integers, only those integers that lie within the specified range.

Unless you specify otherwise, the computer stores the value of all numeric variables in double-precision format. To specify that the value of a numeric variable is to be stored in single- or integer-precision format, you must follow the variable name with a *type declaration character*. The following table lists the type declaration characters used by BASIC. [Note the appearance of the familiar dollar (\$) sign, used to indicate a string variable.]

Variable Type	Declaration Character
String	\$
Double precision	#
Single precision	!
Integer precision	%

Because the default variable type can be changed, as described later in this lesson, BASIC provides the # character for explicitly declaring a variable to be double precision.

The single-precision declaration character is the exclamation point (!). To illustrate the effect of this character, enter the following example.

```
S!=10/7
Ok
PRINT S!
1.42857
Ok
```

The computer stores the value of the repeating decimal to six significant places only.

The integer-precision declaration character is the percent sign (%). The following example illustrates the effect of this character.

```
I%=10/7
Ok
PRINT I%
1
Ok
```

The computer converts the repeating decimal to integer precision by simply truncating (dropping) the digits to the right of the decimal point. It performs no rounding operations. If you attempt to assign a value smaller than -32768 or larger than 32767 to an integer-precision variable, an `?OV Error` (overflow) condition occurs.

The variable declaration characters differentiate what would otherwise be identical variable names. Thus,

T (or T#)
T!
T%
T\$

are all different variables, as illustrated by the following example.

```
T$="100/22="
Ok
T=100/22
Ok
T!=100/22
Ok
T%=100/22
Ok
PRINT T$;T;T!;T%
100/22= 4.5454545454545 4.54545 4
Ok
```

Since variable declaration characters seem to complicate what is already an intricate subject, you may be wondering: Why not use double-precision for all numeric variables and avoid the difficulties introduced by trying to remember which numeric variable type to use for a given application?

The answer is: to save time and memory. Double precision, because of the number of significant digits maintained, takes more time for the computer to process and requires more storage space in memory than does single precision; and single precision requires more time and memory space than is required by integer precision. Thus, when memory space or speed is important, it is advantageous to

use the minimal level of precision that is acceptable for an application.

Technical Note: Double-precision numbers require 8 bytes of memory for storage, single-precision numbers require 4 bytes of memory, and integer-precision numbers require 2 bytes of memory.

As an alternative to using a variable declaration character with each individual variable, you can use a *global declaration statement* to reserve sections of the alphabet for various types of variables. The following table lists the global declaration statements used by BASIC.

Statement	Definition
DEFSTR <i>letter range</i>	DEFine STRing
DEFDBL <i>letter range</i>	DEFine DouBLe precision
DEFSNG <i>letter range</i>	DEFine SiNGle precision
DEFINT <i>letter range</i>	DEFine INTeger precision

The *letter range* parameter can consist of one or more single letters separated by commas, or a range of letters specified by a beginning letter, a hyphen, and an ending letter.

Examples of valid declaration statements are shown below.

Statement	Meaning
DEFSTR A	Defines all variables beginning with the letter A as string variables, even though they may not be followed by a \$ sign
DEFDBL X-Y	Defines all variables beginning with the letters X through Z, inclusive, as double precision
DEFSNG S,T,W-Y	Defines all variables beginning with the letters S, T, and W through Y as single precision
DEFINT A-Z	Defines all variables as integer precision (in effect, changing the default variable type to integer precision)

Global declaration statements are normally placed at the beginning of a program, so that the declaration statement will be executed before the affected variable name is used. If an individual variable has a variable declaration character in its name, that declaration overrules any global declaration statement affecting that variable. If conflicting declaration statements are executed, BASIC uses the last executed statement to define a variable's type.

SUMMARY

BASIC provides three levels of accuracy for numeric variables: double precision, single precision, and integer precision. Double precision restricts numeric accuracy to 14 significant digits, single precision restricts accuracy to 6 significant digits, and integer precision limits numeric values to whole numbers in the range -32768 to 32767, inclusive.

By default, the computer stores the value of all numeric variables in double-precision format. To specify another format, either (a) follow the variable name with a *type declaration character* or (b) use a *global declaration character* to specify the type of all variables that lie within a letter range. The following table lists the declaration characters and statements used by BASIC.

Variable Type	Declaration Character	Global Declaration Statement
String	\$	DEFSTR <i>letter range</i>
Double precision	#	DEFDBL <i>letter range</i>
Single precision	!	DEFSNG <i>letter range</i>
Integer precision	%	DEFINT <i>letter range</i>

Notice that the \$ sign that indicates a string variable is only one of four type declaration characters, and that BASIC provides a global declaration statement for defining a range of variables as string variables.

Variable declaration characters differentiate what are otherwise identical variable names. Thus, V#, V!, V%, and V\$ are unique variable names.

The *letter range* parameter of a global declaration statement can be a single letter; two or more letters separated by commas; a range of letters specified by a beginning letter, a hyphen, and an ending letter; or any combination of these.

Global declaration statements are normally placed at the beginning of a program, so that the declaration statement will be executed before the affected variable name is used. If an individual variable has a variable declaration character in its name, that declaration overrules any global declaration statement affecting the variable. If conflicting declaration statements are executed, BASIC uses the last executed statement to define a variable's type.

Simulating Chance Occurrences

Your computer has a higher-order function which permits the computer to simulate chance or random occurrences. This function creates a series of numbers which appear to be random and have the same statistical properties as randomly generated numbers. Because these numbers are generated by a computational procedure, however, they are referred to more accurately as *pseudorandom numbers*.

The general form of the pseudorandom number function is

RND (*numeric expression*)

The **RND** function returns a number in the range from 0 through 1, where 0 is a possible value but 1 is not, each time the function is executed. A program illustrating **RND** is shown below.

```

NEW
100 PRINT RND(9)
110 GOTO 100

```

RUN the program.

```

RUN
.59521943994623
.10658628050158
.76597651772823
.57756392935958
.73474759503023
.18426812909758

```

(Press **CTRL** **C** to stop program execution.)

Although it appears that **RND** does produce numbers in the range described, you are probably wondering how these numbers can be considered random. After all, didn't your program produce the same numbers as those printed in this book?

The result is not really contradictory and can be explained as follows. The computer begins its generation of pseudorandom numbers with an initial value known as a *seed value*. If you use a positive argument with the **RND** function, the computer always uses the same initial seed value. Consequently, the same series of pseudorandom numbers is generated each time the program is run.

To create a different sequence of random numbers, you must instruct the computer to use a different seed value when it begins generating the pseudorandom numbers. If you use a negative argument with the **RND** function, the computer uses the value of the argument as the seed value. As an experiment, make the following change to line 100 and **RUN** the program again.

```

100 PRINT RND(-9)
RUN
.24389820420821
.24389820420821
.24389820420821

```

```
.24389820420821
.24389820420821
```

(Press **CTRL** **C** to stop program execution.)

The computer generates a different initial pseudorandom number. But because it uses the same seed value (-9) every time the function is executed, it generates the same pseudorandom number over and over.

The secret to creating a sequence of pseudorandom numbers that vary from the values generated by the first example is to execute the **RND** function once with a negative argument (to set the seed value), and then use a positive argument to generate the series. The following program illustrates this technique. Notice that the **RND** function used to set the seed value is given a line number outside the loop used to generate the series of pseudorandom numbers.

```
NEW
90 A=RND (-9)
100 PRINT RND(9)
110 GOTO 100
```

When you **RUN** this program, you will get a series of pseudorandom numbers that differ from the sequence generated by the first example, although the same sequence is generated every time you **RUN** the program.

To create an unpredictable sequence of pseudorandom numbers, you must use a different negative seed value each time the program is run. Although this could be accomplished by manually changing the seed value every time the program is executed, it is more convenient to let the program perform this task. The following expression uses the **TIMES** statement to calculate a negative number that can be used as an unpredictable seed value. (This expression uses several functions not yet discussed to find the current seconds' setting of the computer's clock, which is then negated to create a negative seed value. For a discussion of these functions, refer to Lessons 30, 35, and 40.)

```
A=RND(-VAL(RIGHT$(TIME$,2))-1)
```

Add this randomizing feature to the program developed earlier by changing line 90 to

```
90 A=RND(-VAL(RIGHT$(TIME$,2))-1)
```

RUN and break the program several times. You will get a different series of pseudorandom numbers each time you execute the program.

The reproducibility of the pseudorandom number sequence is actually an asset, as it allows complicated programs that use random numbers to be developed without worrying about whether operational differences are the result of programming errors or changes in the sequence of random numbers. Since **RND** can be trusted to generate the same series of numbers when any positive argument is used, any differences in results are attributable to programming problems. When a program is fully error free, the randomizing expression can be added to create unpredictable random number sequences.

If a zero argument is used with the **RND** function, the function returns the last pseudorandom number generated.

The following program illustrates how **RND** can be put to a more practical use. This program uses **RND** to simulate the tossing of a coin.

```
NEW
100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 R=RND(1)
120 IF R<.5 THEN PRINT "heads"
130 IF R=>.5 THEN PRINT "tails"
140 GOTO 110
```

The program begins with the randomizing expression to ensure an unpredictable sequence of pseudorandom numbers. Then the program generates a random number. If the random number is less than .5, the message "heads" is displayed. If it is greater than or equal to .5, "tails" is displayed. (The nature of the pseudorandom

number generating process of the computer is such that the likelihood of a pseudorandom number being in either of these categories is exactly equal.)

RUN the program and observe the results. The sequence of heads and tails that are displayed should appear as random as actually flipping a coin.

```

RUN
heads
                                tails
                                tails
                                tails
heads

```

(Press **CTRL** **C** to stop program execution.)

The sequence produced by your computer will probably differ from the sample shown above.

The **RND** function is ideal for programs such as the heads or tails example where it does not matter that the random numbers are decimal values. For some applications, however, it is desirable to have random numbers that lie within a given range. To generate a range of random integers (whole numbers) between 1 and a specific upper limit, use the expression

$$N = \text{INT}(L * \text{RND}(1) + 1)$$

where N = the generated random integer and L = the selected upper limit. For example, if $L = 100$, then N will equal a random integer between 1 and 100. If $L = 12$, then N will equal a random integer between 1 and 12. (The randomizing expression must still be used to create an unpredictable sequence of numbers.)

The "random integer" expression works as follows. The pseudorandom number generated by **RND** (which ranges from 0 to .999999999999999) is multiplied by L to obtain a value between 0 and $L + .999999999999999$. 1 is then added to this value before the **INT** function is performed to force values of $L + .999999999999999$ to round to L .

The following program illustrates how the random integer expression can be used to simulate the roll of a pair of dice.

```

NEW
100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 D1=INT(6*RND(1)+1)
120 D2=INT(6*RND(1)+1)
130 PRINT D1,D2
140 GOTO 110

```

The preceding programs illustrated how pseudorandom numbers can be used to simulate events that are truly random such as flipping a coin or rolling a pair of dice. Applications such as these are the basis for many computer games. Pseudorandom numbers are also useful for combining numeric or textual information in unpredictable ways. For example, a program that uses random numbers to provide drill practice for learning the multiplication tables is shown below.

```

NEW
100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 N1=INT(9*RND(1)+1)
120 N2=INT(9*RND(1)+1)
130 PRINT N1;"*";N2;"=" ";
140 INPUT ANSWER
150 IF ANSWER=N1*N2 THEN 180
160 PRINT "WRONG. TRY AGAIN."
170 GOTO 130
180 PRINT "VERY GOOD!"
190 GOTO 110

```

Observe that the pending display state created by the final semicolon in line 130 causes the **INPUT** statement in line 140 to display its "?" prompt on the same display line as the multiplication problem.

A sample **RUN** of the program is shown below.

```

RUN
  4 * 6 = ? 24
VERY GOOD!
  8 * 7 = ? 48

```

WRONG. TRY AGAIN.

8 * 7 = ? 56

VERY GOOD!

8 * 1 = ? 8

VERY GOOD!

6 * 6 = ?

(Press **CTRL** **C** to stop program execution.)

As written, this program provides drill practice for problems up to 9×9 . The upper range of the program can be altered by changing the value of *L*. For example, using a value of 11 provides practice up to 11×11 .

You can alter the type of drill practice by changing the mathematical operator in line 150 and the prompt in line 130. For example, changing these lines to

```
130 PRINT N1;"+";N2;"= ";
150 IF ANSWER=N1+N2 THEN 180
```

will generate addition problems.

SUMMARY

The higher-order function used to generate pseudorandom numbers is

RND (*numeric expression*)

The **RND** function returns a pseudorandom number in the range 0 to .999999999999999. If the value of *numeric expression* is positive, the same sequence of pseudorandom numbers is created every time a program is run. If the value of *numeric expression* is negative, that value is used as a seed value to create the pseudorandom number. If the value of *numeric expression* is zero, the function returns the last number generated by the function.

Due to the computational nature of the function, the same seed value generates the same sequence of random numbers every time

a program is run. This reproducibility is an advantage when developing a program, but is usually undesirable in a finished program. To create an unpredictable sequence of pseudorandom numbers, it is necessary to use a different negative seed value each time a program is run. The following expression calculates a number based on the time setting of the computer that can be used as an unpredictable seed value, to ensure an unpredictable series of pseudorandom numbers.

$$A = (-\text{VAL}(\text{RIGHT}(\text{TIME}, 2)) - 1)$$

Some applications require randomly selected integers lying within a given range. To generate whole numbers between 1 and a specific upper limit, use the expression

$$N = \text{INT}(L * \text{RND}(1) + 1)$$

where N = the generated random integer and L = the selected upper limit.

REVIEW TEST 3

1. What is meant by the phrase "levels of precedence"? What is the purpose of using parentheses in a calculation?
2. How can the exponentiation operation be used to compute the square root of a number? The fifth root of a number?
3. Which operation is performed first when you enter the sequence $-7*2$ into the computer?
4. Write one-line programs to display the results of the following problems.

(a) $\sqrt{12 + 77/3.14159}$

(b) $118.56 \times 2.022^{2+3.8}$

(c) $\frac{1}{9} + \frac{1}{7} + \frac{1}{5} + \frac{1}{3}$

(d) $\frac{142}{117} + \frac{93}{12}$

$$\frac{16}{40}$$

(e) $\frac{\sin 206 - \tan 94}{\cos 1.71}$

5. What is the “argument” of a function? What is meant by stating that an argument can be any valid *numeric expression*?
6. Write a program that computes the square root of any positive number that is input but displays “INVALID” for any negative input. Use the **SGN** function to make the test.
7. What is the purpose of the “randomizing expression”?
8. What is the largest pseudorandom number generated by **RND**? The smallest?
9. Using **RND**, write a program to generate and sum 1000 random numbers. What is the sum likely to be?
10. Write a program that displays integer random numbers between 100 and 199, inclusive.

FOR TO NEXT Looping

In Lesson 8 you learned how to control the number of times that a loop was executed by establishing a “counting variable” and increasing it by 1 each time the loop was executed. An **IF THEN** statement was used to determine when to stop executing the loop.

Because looping is such an important part of programming, BASIC provides two special statements for this purpose—the **FOR TO** and **NEXT** statements. The **FOR TO** and **NEXT** statements have the general form.

```
FOR control variable=initial value TO limit  
.  
.  
.  
NEXT control variable
```

(the statements to be executed in the loop go here)

where *control variable* is the statement’s “counting variable,” *initial*

value is the starting value of the control variable, and *limit* is the termination value. The instructions to be executed in the loop are placed between the **FOR TO** and **NEXT** statements.

The function of the **FOR TO** statement is to specify the *control variable* and define the starting and ending values of the variable. The **FOR TO** statement also marks the physical beginning of the loop. The function of the **NEXT** statement is to increase the value of the *control variable* and then decide if execution is to be sent back to the beginning of the loop or allowed to continue with the statement following the loop. If the *control variable* is *less than or equal to* the limit value, execution is sent back to the first statement following the **FOR TO** statement. If the *control variable* is *greater than* the limit value, execution continues with the first statement following the **NEXT** statement.

An example of a **FOR TO NEXT** loop is shown below.

```
NEW
100 FOR COUNT=1 TO 8
110 PRINT COUNT
120 NEXT COUNT
130 PRINT "Loop done"
```

RUN the program and observe the results.

```
RUN
1
2
3
4
5
6
7
8
Loop done
```

The **FOR TO** and **NEXT** statements cause the computer to repeat the loop eight times.

When the **FOR TO** statement is executed, the computer sets the value of COUNT to 1. Execution then proceeds with line 110. When the **NEXT** statement in line 120 is executed, the computer adds 1 to the value of COUNT and checks whether it is time to end the loop. Since COUNT is only equal to 2 at this point, execution is transferred back to line 110.

The program repeats this looping procedure seven more times. At the end of the eighth execution of the loop, the **NEXT** statement increases the value of COUNT to 9, which *exceeds* the limit specified by the **FOR TO** statement. As a result the computer does not transfer execution back to line 110, but allows the program to continue with line 130. If you check the value of COUNT after executing the program, you will find it to be 9.

```
PRINT COUNT
```

```
9
```

```
Ok
```

The *control variable* is larger than the *limit*.

The values selected for *initial value* and *limit* determine the number of times the loop is executed. You can calculate this number using the formula

$$\text{number of loops} = \text{limit} + 1 - \text{initial value}$$

The value of *limit* is increased by 1 because *limit* must be *exceeded* before the loop is exited.

You can use any values you want for *initial value* and *limit* as long as they cause the computer to execute the loop the desired number of times. For example, change line 100 of this program to

```
100 FOR COUNT=1001 TO 1008
```

and **RUN** the program again.

```
RUN
```

```
1001
```

```
1002
```

```

1003
1004
1005
1006
1007
1008
Loop done

```

The values displayed for COUNT are different, of course, but the number of times that the loop is executed is the same ($1008 + 1 - 1001 = 8$).

As you can see, **FOR TO NEXT** looping is very similar to looping with a counting variable and an **IF THEN** statement. The major difference is that the **NEXT** statement combines the functions of increasing the counting variable, testing if the termination value has been reached, and transferring execution to the beginning of the loop if the terminating value has not been reached. Since they are so similar, you may be wondering which technique to use. Although there are instances in which a counting variable and an **IF THEN** statement is the better choice, as a general rule you should use **FOR TO** and **NEXT** statements to create loops *when you know how many times the loop is to be executed*.

The next example illustrates how the averaging program developed in Lesson 8 can be modified to use a **FOR TO NEXT** loop.

```

NEW
100 SUM=0
110 INPUT "Number of entries";N
120 FOR COUNT=1 TO N
130 INPUT "Entry";ENTRY
140 SUM=SUM+ENTRY
150 NEXT COUNT
160 AVERAGE=SUM/N
170 PRINT "Average =";AVERAGE

```

In this program, a variable is used to specify the *limit*. You can use any valid numeric expression as an *initial value* or *limit*. If a

variable is used, its value is not changed by the execution of the **FOR TO** statement. Thus, you can be certain that N retains the correct value for calculating the average in line 150, even after it has been used to define the **FOR TO NEXT** limit.

```

RUN
Number of entries? 5
Entry? 11
Entry? 22
Entry? 33
Entry? 44
Entry? 55
Average = 33
Ok

```

The *control variable* of a **FOR NEXT** loop can do more than just count the number of times a loop is executed. When convenient, the *control variable* can also be used in any calculations performed by the loop. If used in a calculation, however, it should not be changed unless you intend to change the number of times the loop is executed.

The following program provides a good example of an application where it is desirable to use a *control variable* in a calculation. This program computes the factorial of any whole number up to 49 that you enter (the factorial of values larger than 49 exceed the computational limit of the computer). The factorial of a whole number is defined as the product of that number and all positive whole numbers less than the number. For example, the factorial of $7 = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$.

```

NEW
100 FACT=1
110 INPUT "Enter number";N
120 IF N>49 THEN 110
130 FOR COUNT=1 TO N
140 FACT=FACT*COUNT
150 NEXT COUNT
160 PRINT "The factorial is";FACT

```

To compute the factorial, the program uses a loop to multiply the variable **FACT** by every whole number between 1 and the number entered.

A sample **RUN** of the program is shown below.

```

RUN
Enter number? 12
The factorial is 479001600
Ok
RUN
Enter number? 45
The factorial is 1.1962222086547E+56
Ok

```

The factorial of 45 is too large to display in regular format, so it is displayed in scientific notation.

It is common in creating tables of numbers to use the *control variable* of a loop in the calculation. The following program illustrates this technique.

```

NEW
100 KARAT=4.167
110 FOR COUNT=1 TO 24
120 PERCENTGOLD=KARAT*COUNT
130 PRINT COUNT;"karats =";PERCENTGOLD;"% gold"
150 NEXT COUNT

```

This program computes the percentage of gold in karat values 1 through 24. The calculations are performed by multiplying the value of the loop counter by 4.167 (the approximate percentage of gold in 1 karat). Since pure gold is 24 karats, the limit of the *control variable* is 24. A partial program run is shown below.

```

RUN
1 karats = 4.167% gold
2 karats = 8.334% gold
3 karats = 12.501% gold
4 karats = 16.668% gold

```



```

.
.
.
22 karats = 91.674% gold
23 karats = 95.841% gold
24 karats = 100.008% gold
Ok

```

Here is another example where the *control variable* of a **FOR NEXT** loop is used in a calculation. This program calculates the amount of money generated by interest rates ranging from 10 to 15 percent compounded monthly for 12 months. The formula for this calculation is

$$FV = P * (1 + I)^N$$

where FV is the future value of the money, P is the principal or amount deposited, I is the interest rate per compounding period, and N is the number of compounding periods. In this example, N equals 12 since the principal is compounded monthly for 1 year and I is the value of the *control variable* divided by 100 (division by 100 is necessary to convert I to a decimal value).

```

NEW
100 INPUT "Principal";P
110 FOR COUNT=10 TO 15
120 I=COUNT/100
130 FV=P*(1+I/12) ^ 12
140 FV=INT(FV*100+.5)/100
150 PRINT "Value at";COUNT;"% =" ;FV;"dollars"
160 NEXT COUNT

```

The purpose of the calculation in line 140 is to round the value of FV to two decimal places. Refer to Lesson 13 for a discussion of this formula.

A sample **RUN** for a deposit of \$1425.17 is shown below.

```

RUN
Principal? 1425.17
Value at 10% = 1574.4 dollars

```

```
Value at 11% = 1590.09 dollars
Value at 12% = 1605.92 dollars
Value at 13% = 1621.89 dollars
Value at 14% = 1638.01 dollars
Value at 15% = 1654.27 dollars
Ok
```

SUMMARY

The function of the **FOR TO** statement in a **FOR TO NEXT** loop is to specify the counting variable of the loop (known as the *control variable*) and define the starting and ending values of the variable, known as the *initial value* and *limit*, respectively. The **FOR TO** statement also marks the physical beginning of the loop. The value of the *initial value* and *limit* can be defined by any valid numeric expression.

The function of the **NEXT** statement is to increase the value of the *control variable* and decide if execution is to be sent back to the beginning of the loop or allowed to continue with the statement following the loop. If the *control variable* is less than or equal to the *limit* value, execution is sent back to the first statement following the **FOR TO** statement. If the *control variable* is greater than the *limit* value, execution continues with the first statement following the **NEXT** statement.

The *control variable* of a **FOR NEXT** loop can be used in calculations within the loop when desirable. If used in a calculation, the *control variable* should not be changed unless you intend to change the number of times the loop is executed.

It is recommended that you use **FOR NEXT** loops whenever you know how many times the loop is to be executed. The advantages of **FOR NEXT** loops are:

1. **FOR NEXT** loops are shorter and more efficient than loops that use a counting variable and an **IF THEN** statement.

2. **FOR NEXT** loops are easier to write and reduce the chances of making a programming mistake.
3. A program that uses a **FOR NEXT** loop is easier to understand than a program that uses an **IF THEN** statement and a counting variable to control a loop.

REM Statements

As you know by now, it is not always easy to understand how a program works. For this reason, BASIC provides a method of entering remarks or explanatory information directly into a program. The statement used for this purpose is the **REM** statement.

The **REM** statement instructs the computer to ignore completely the rest of the program line. As a result, you can place any information you want following a **REM** statement. You do not have to avoid using BASIC words or symbols in your remarks.

NEW

```
100 REM *****
110 REM This program consists
120 REM entirely of REM statements.
130 REM Since it contains
140 REM no active statements,
150 REM nothing happens
```

```
160 REM when it is executed.  
170 REM *****
```

The information that you enter following a **REM** statement is unaltered. The computer does not convert lowercase letters to uppercase letters as it does for program statements.

It is good programming practice to use **REM** statements in your programs. Good remarks make a program much more understandable for others. They can also prevent you from wasting time trying to figure out what some part of your program does when you want to modify a program several weeks or months after writing it. Many programmers put their name, the name of the program, and the date that the program was written in **REM** statements at the beginning of every program they write. For longer programs, you may want to put information such as the variables used, the purpose of each variable, and the function of the major parts of the program in **REM** statements.

The following example illustrates the use of remarks in a program. Because of the many **REM** statements, you should need no additional explanation of the program's operation.

```
NEW  
100 REM Balance checkbook program  
110 REM N1=number of deposits  
120 REM N2=number of withdrawals (checks)  
130 INPUT "Starting balance";BALANCE  
140 INPUT "Number of deposits";N1  
150 INPUT "Number of withdrawals";N2  
160 REM The next line tests for no deposits  
170 IF N1=<0 THEN 230  
180 REM ** Begin deposit loop  
190 FOR COUNT=1 TO N1  
200 INPUT "Deposit amount";DEP  
210 BALANCE=BALANCE+DEP  
220 NEXT COUNT  
230 REM The next line tests for no withdrawals  
240 IF N2=<0 THEN 300
```

```

250 REM ** Begin withdrawal loop
260 FOR COUNT=1 TO N2
270 INPUT "Withdrawal amount";WITHDRAWAL
280 BALANCE=BALANCE—WITHDRAWAL
290 NEXT COUNT
300 PRINT "Your new balance is $";BALANCE

```

Notice that although **REM** statements do not perform any action during the running of a program, you can still transfer program execution to line numbers containing **REM** statements.

A sample **RUN** of the checkbook program is shown below.

```

RUN
Starting balance? 412.19
Number of deposits? 1
Number of withdrawals? 5
Deposit amount? 10.42
Withdrawal amount? 125
Withdrawal amount? 79.51
Withdrawal amount? 9.95
Withdrawal amount? 12.98
Withdrawal amount? 5
Your new balance is $ 190.17
Ok

```

As an alternative to the word **REM**, you can use an apostrophe (') to indicate a remark. The apostrophe serves the same function as **REM** and has the additional advantage that it can be placed at the end of a regular program line. An example of a program line that uses the apostrophe for a remark is shown below.

```

170 IF N=<0 THEN 230 'Test for no deposits

```

SUMMARY

The function of the **REM** statement is to allow descriptive remarks to be placed within a program. The **REM** statement instructs the computer to ignore the remainder of the program line.

The apostrophe (') is an alternative to **REM** that can be used at the end of a regular program line.

Storing DATA in a Program

Many programming applications require the manipulation of large quantities of information. For example, a payroll program might store the following information for each employee of a company: name, address, pay rate, marital status, and number of tax exemptions. This information could be placed in a program by using assignment statements, as shown below.

NEW

100 'Payroll information

110 N1\$="Brock"

'Name

120 A1\$="7000 Semiconductor Drive"

'Address

130 R1=6.03

'Pay rate

140 S1\$="married"

'Marital status

150 E1=2

'Exemptions

160 N2\$="Clark"

'Name

170 A2\$="4116 Memory Lane"	'Address
180 R2=4.75	'Pay rate
190 S2\$="married"	'Marital status
200 E2=2	'Exemptions
.	
.	
.	

But as you can see, if the company has many employees, this method of storing information requires a large number of variables and creates a program that is long and cumbersome. Fortunately, BASIC provides a more efficient method of storing information in a program. The basis of this alternative is the **DATA** statement.

The **DATA** statement has the general form

DATA *data list*

where *data list* is a list of string and/or numeric constants separated by commas.

The sole function of the **DATA** statement is to store information in a concise and organized manner. An example of the statement is shown below.

```
110 DATA Brock,7000 Semiconductor Drive,6.03,married,2
120 DATA Clark,4116 Memory Lane,5.32,married,2
```

Note that string information can be stored without quotation marks and that strings and numbers can be intermixed.

Technical Note: The use of quotation marks around string information in **DATA** statements is optional except when a string contains leading spaces, trailing spaces, or commas. If you want to put leading spaces, trailing spaces, or commas in a **DATA** item, you must enclose the item in quotation marks.

The information in all the **DATA** statements of a program is considered to be one large list that begins with the first item of the first

DATA statement and ends with the last item of the last **DATA** statement. Therefore, the number of items following a particular **DATA** statement is unimportant—what is important is the order in which items are listed. Thus

```
110 DATA Brock,7000 Semiconductor Drive,6.03,married,2
```

and

```
110 DATA Brock
120 DATA 7000 Semiconductor Drive
130 DATA 6.03
140 DATA married
150 DATA 2
```

form identical lists of data as far as the computer is concerned. Of course, the first example stores the information in fewer program lines.

To access the information stored in a **DATA** statement, you must use the **READ** statement. The general form of the **READ** statement is

```
READ variable list
```

where *variable list* is a list of string and/or numeric variables separated by commas.

The **READ** statement assigns the information in a *data list* to the variables in a *variable list*. The **DATA** items are always assigned in the precise order in which they appear in the program. Thus, the first **READ** statement assigns the first **DATA** items to the variables in its *variable list*, the second **READ** statement assigns the next **DATA** items to the variables in its *variable list*, and so on for each **READ** statement that is executed. The computer keeps an exact count of the **DATA** items that have been assigned already and ensures that the next **READ** statement always begins with the first unread **DATA** item.

The following example illustrates how this works.

NEW

```

100 'Read data list example
110 DATA Brock,7000 Semiconductor Drive,6.03,married,2
120 DATA Clark,4116 Memory Lane,5.32,married,2
130 DATA Dever,Software Circle,4.75,single,2
140 DATA Riddle,1024 Byte Road,4.15,married,2
150 DATA Smith,911 Silicon Park,7.75,single,1
160 DATA Stewart,8085 Processor Boulevard,6.95,single,1
170 FOR COUNT=1 TO 6
180 READ NA$,ADDRESS$,RATE,STATUS$,TAXEXMPT
190 PRINT NA$,STATUS$
200 NEXT COUNT
210 GOTO 210

```

A sample **RUN** of the program is shown below.

```

RUN
Brock      married
Clark      married
Dever      single
Riddle     married
Smith      single
Stewart    single

```

(Press **CTRL** **C** to stop program execution.)

This program places the **READ** statement in a loop and reads the data items into the same variable names each time the loop is executed. Although the information could be read into different variable names, one of the advantages of using **DATA** statements is that information does not have to be saved in variables since it has already been permanently stored in the program in the form of **DATA** statements. Observe that the program reads string information into string variables and numeric information into numeric variables. BASIC allows numeric information to be read into either a numeric or string variable, but it permits string information to be read only into a string variable.

Notice also that even though the information that is read into **ADDRESS\$, RATE, and TAXEXMPT** is not used by the program, it is

necessary to read that information in order to get the name and marital status of the next employee.

DATA statements can be placed anywhere that you find convenient in a program. To verify this fact, delete lines 150 and 160 and reenter them at the end of the program.

```

150      (To delete lines 150 and 160)
160
220 DATA Smith,911 Silicon Park,7.75,single,1
230 DATA Stewart,8085 Processor Boulevard,6.95,single,1

```

The new location of the **DATA** statements does not alter the execution of the program.

```

RUN
Brock      married
Clark      married
Dever      single
Riddle     married
Smith      single
Stewart    single

```

(Press **CTRL** **C** to stop program execution.)

Because **DATA** statements can appear anywhere in a program, different programmers develop different preferences for their location. Some prefer to place them at the beginning of a program, some at the end of a program, and some scatter them throughout a program.

When using **DATA** statements, you must be careful that your program does not attempt to read more data items than exist in the program. If a **READ** statement is executed after the last data item has been read, an **END Error** (Out of Data) condition occurs.

To avoid data errors, you must prevent your program from reading past the last data item. This can often be accomplished by manually counting the number of data items and using that value in a **FOR NEXT** loop. In some programming applications, however, you may

want to add or subtract data each time the program is used. In such cases, a common technique is to store a "dummy" value as the last data item in the program. The program can then test the value of data items as they are read to determine when the end of the data list is reached. The example that follows illustrates this programming technique.

```

NEW
100 'Grocery list program
110 READ DESC$,QUANTITY,UNIT$
120 IF QUANTITY=-1 THEN END
130 PRINT QUANTITY;UNIT$;" ";DESC$
140 GOTO 110
150 DATA eggs,2,dozen,milk,1,quart
160 DATA buns,3,packages,lettuce,2,heads
170 DATA softdrinks,7,liters
999 DATA xxx,-1,xxx

```

This program displays a list of foods to be bought on the next grocery trip. Since this list will presumably change as purchases are made or supplies are depleted, it is organized so that the last items in the data list are dummy values. There must be three dummy items to correspond to the three variables in the **READ** statement. The computer can test for a quantity of -1 to determine when the last data item has been read.

A sample **RUN** is shown below.

```

RUN
 2 dozen eggs
 1 quart milk
 3 packages buns
 2 heads lettuce
 7 liters softdrinks
Ok

```

As new supplies are needed, they can be added to the list by simply entering a new **DATA** statement with a line number less than 999. To remove a food from the list, delete the line containing the **DATA**

statement where it is listed or remove the food from the **DATA** statement. Just be certain to keep the string and numeric data in the correct order. The program will automatically adjust itself to the number of data items present.

```
150      (To delete line 150)
190 DATA cheese,2,pounds
```

A sample **RUN** is shown below.

```
RUN
  3 packages buns
  2 heads lettuce
  7 liters soft drinks
  2 pounds cheese
```

It is often desirable to read a *data list* more than once during the execution of a program. The **RESTORE** statement allows a program to read a *data list* over and over.

The general form of the **RESTORE** statement is

```
RESTORE line number
```

The **RESTORE** statement specifies the line number where the next **READ** statement that is executed is to begin reading data. The line number specified by the **RESTORE** statement does not have to contain a **DATA** statement. If there is no **DATA** statement on that line, the **READ** statement will begin with the first **DATA** statement after the specified line number. If no line number is given with the **RESTORE** statement, the next **READ** statement will begin with the first item in the data list.

The following program illustrates the use of the **RESTORE** statement.

```
NEW
100 'Search for inventory item
110 INPUT "Enter ID number";ID
120 READ N,QUANTITY
```

```

130 IF N=-1 THEN 170 'Test for end of data
140 IF N<>ID THEN 120 'Test for id number
150 PRINT QUANTITY;ID;"in stock"
160 GOTO 180 'Skip next line
170 PRINT ID;"not found"
180 RESTORE 100 'Reset data pointer
190 GOTO 110 'loop
200 DATA 1001,7,1011,23,1003,3
210 DATA 1022,12,1009,1,1012,0
220 DATA 1033,4
999 DATA -1,-1

```

This program searches through the information stored in a *data list* for stock ID numbers. If the ID number is found, the quantity on hand is displayed. If the ID number is not found, the message "not found" is displayed. Since it is necessary to search through the *data list* from the beginning each time a new ID is entered, a **RESTORE** 100 statement is executed at the end of every search. The **RESTORE** 100 ensures that the *data list* is read from the beginning.

A sample **RUN** is shown below.

```

RUN
Enter ID number? 1017
1017 not found
Enter ID number? 1022
12 1022 in stock
Enter ID number?

```

(Press **CTRL** **C** to stop program execution.)

SUMMARY

The **DATA** statement permits numeric and string information to be stored in a program without the requirement to assign it to variables. The items of information stored in a **DATA** statement must be separated by commas. It is not necessary to place quotation marks

around string data items unless the items include leading spaces, trailing spaces, or commas.

The **DATA** statement is not executed when a program is **RUN**. To access the information stored by **DATA** statements, it is necessary to use the **READ** statement. The **READ** statement assigns the information in **DATA** statements to variables in the **READ** statement. **DATA** statements can be placed wherever convenient in a program. It is not the location or number of **DATA** statements that is important, but the order in which information is listed in the statements. The **DATA** information is always assigned in the precise order in which it appears in the program.

BASIC allows numeric data to be read into either numeric or string variables, but it permits string data to be read only into string variables.

The **RESTORE** statement can be used to allow a program to reread data items. The general form of the statement is

RESTORE *line number*

where *line number* indicates the location in the program where the next **READ** statement is to begin reading. If the specified line number does not contain a **DATA** statement, reading begins with the first **DATA** statement found after that line number. If no line number is given, reading begins with the first item in the data list.

Arrays and Subscripts

The examples that you have studied so far have used what are known as *simple variables* to store information. Simple variables can be either numeric or string, but have the characteristic that they can be assigned only one value at a time. By contrast, an *array* variable can be assigned many values at a time. The different values assigned to an array variable are called *elements* and are kept separate by giving each value a unique *subscript*.

An example of an array variable is shown below.

GRADE(3)=95

↑ ↙ ↘

Array name Subscript

This sequence assigns the value 95 to element 3 of an array named GRADE. The subscript that identifies the element is placed in parentheses immediately following the array's name.

Other array elements can be assigned just as easily.

```
GRADE(1)=88
GRADE(5)=100
```

The GRADE array as it is now defined is illustrated below. Notice that the first element is GRADE(0). *The first element of an array variable is always element 0.* Since you have not assigned values for elements 0, 2, and 4, they are shown as having zero values.

```
GRADE(0)=0
GRADE(1)=88
GRADE(2)=0
GRADE(3)=95
GRADE(4)=0
GRADE(5)=100
```

String arrays can be created by using a dollar sign as the last character of the array name. Thus,

```
WRITER$(1)="Shakespeare"
```

assigns the string value "Shakespeare" to element 1 of a string array named WRITER\$.

Once a value has been assigned to a particular array element, you can use that element just as any variable as long as you use the correct subscript when referring to it.

```
GRADE(4)=52
Ok
PRINT GRADE(4)*2
104
Ok
WRITER$(0)="Miller"
Ok
PRINT WRITER$(0)
Miller
Ok
```

It is the subscript following a variable name that identifies the variable as an array variable. If you drop the subscript, you are no longer referring to an array. Thus,

```
GRADE #
GRADE!
GRADE%
GRADE$
GRADE #(2)
GRADE!(2)
GRADE$(2)
GRADE%(2)
```

are entirely different variables. The first four are simple variables and the last four are elements of array variables.

The rules for forming array names are the same as for forming simple variable names with the added restriction that the array element be specified by a subscript enclosed in parentheses. The type declaration characters and global declaration statements (discussed in Lesson 14) apply to array variables just as they do for simple variables. For example, if an array is to be restricted to integer precision, you must declare the array to be an integer array using the % declaration character or the **DEFINT** statement. (As with simple variables, the default variable type for undeclared array variables is double precision.) The following are examples of valid array names:

```
X%(2)
PAYROLL!(5)
PRICE(4)
YEAR$(1)
L$(7)
```

The first three examples are numeric arrays (declared to be integer, single-precision, and double-precision arrays, respectively). The last two examples are string arrays.

In the examples given so far, the number of the array element is identified by a constant. If you always had to use constants to specify

an array element, array variables would be no more useful than simple variables. The strength of array variables is that subscripts can be identified by other variables. The following program provides an example of this feature.

```
NEW
100 ' Array demonstration program
110 FOR N=0 TO 4
120 INPUT "Enter sample number";SAMPLE(N)
130 NEXT N
140 PRINT "The numbers entered were:"
150 FOR N=0 TO 4
160 PRINT "Element";N;"=";SAMPLE(N)
170 NEXT N
```

A sample **RUN** of the program appears below.

```
RUN
Enter sample number? 100
Enter sample number? 101
Enter sample number? 102
Enter sample number? 103
Enter sample number? 104
The numbers entered were:
Element 0=100
Element 1=101
Element 2=102
Element 3=103
Element 4=104
Ok
```

This program accepts the numbers that you enter and stores them in elements 0 through 4 of an array named **SAMPLE**. After you have entered the five numbers, the program displays them one at a time, showing the array element in which they were stored.

The use of an array variable in this example makes it possible to enter and display the sample numbers using a loop, since the array element in which the numbers are stored is changed by simply changing the value of the subscript variable. If simple numeric varia-

bles were used instead of an array variable, the input part of the program, for example, would require a sequence such as

```
110 INPUT "Enter sample number";N0
120 INPUT "Enter sample number";N1
130 INPUT "Enter sample number";N2
140 INPUT "Enter sample number";N3
150 INPUT "Enter sample number";N4
160 INPUT "Enter sample number";N5
```

The display portion of the program would be equally repetitious.

If you modify the array demonstration program to accept 12 or more sample numbers, a ?BS Error (Bad Subscript) message will be displayed when the program attempts to assign a value to the twelfth element. The reason for this error is that the computer does not allow you to use an array with more than 11 elements unless you have first defined the size of the array.

The size of an array is defined by the **DIM** statement. The **DIM** (for *dimension*) statement has the general form

```
DIM array name(array size)
```

where *array name* is the name of the array variable and *array size* is the number of the largest element of the array that will be used. Since the first element in any array is element 0, the number of elements defined by the **DIM** statement will be the value of *array size* + 1. Several examples of valid **DIM** statements are shown below.

Statement	Meaning
DIM TEST\$(25)	25 is the largest valid element
DIM MILES(55)	55 is the largest valid element
DIM N%(12),P\$(3)	12 is the largest valid element of array N%, 3 is the largest valid element of array P\$

Observe that a single **DIM** statement can define the size of more than one array variable, as long as the different array names are separated by commas.

Although the computer will allow array variables of 11 or fewer elements to be used without requiring a **DIM** statement, it is a good idea to define even small arrays. If you do not define an array variable's size with a **DIM** statement, the computer will automatically reserve enough memory for 11 elements. If you know that fewer than 11 elements are to be used, you can save memory by dimensioning the array for the smaller number of elements.

The following rules must be followed when dimensioning arrays:

1. The **DIM** statement must always be executed before the first use of an array variable in a program.
2. An array can be dimensioned only once during the execution of a program. Thus, the size of an array cannot be changed by a program after it has been defined. If you attempt to redimension an array, a ?DD Error (Doubly Dimensioned array) message is displayed.

An example of a program requiring a **DIM** statement is shown below. This program prompts for the entry of 12 monthly utility bills and stores the values that are entered in an array named BILL. After all the values are entered, the program displays the yearly total, the monthly average, the month of the highest charge, and the month of the lowest charge.

```

NEW
100 ' Monthly utility bill program
110 DIM MO$(12),BILL(12)
120 FOR C=1 TO 12 'Start of input loop
130 READ MO$(C)
140 PRINT "Enter " + MO$(C) + "'s bill";
150 INPUT BILL(C)
160 NEXT C 'End of input loop
170 SUM=0
180 SMALLEST=1
190 LARGEST=1
200 FOR C=1 TO 12 'Start of calculation loop
210 SUM=SUM+BILL(C)
220 IF BILL(C)<BILL(SMALLEST) THEN SMALLEST=C

```

```

230 IF BILL(C) > BILL(LARGEST) THEN LARGEST=C
240 NEXT C           'End of calculation loop
250 PRINT "Yearly total =";SUM
260 PRINT "Average bill =";SUM/12
270 PRINT "Smallest bill was in ";MO$(SMALLEST)
280 PRINT "Largest bill was in ";MO$(LARGEST)
290 DATA January,February,March,April
300 DATA May,June,July,August
310 DATA September,October,November,December

```

A sample program **RUN** is shown below.

```

RUN
Enter January's bill? 46.15
Enter February's bill? 45.4
Enter March's bill? 40.88
Enter April's bill? 38.45
Enter May's bill? 30.07
Enter June's bill? 31.3
Enter July's bill? 39.52
Enter August's bill? 43.17
Enter September's bill? 36
Enter October's bill? 31.62
Enter November's bill? 35.22
Enter December's bill? 38.9
Yearly total = 456.68
Average bill = 38.0566666666667
Smallest bill was in May
Largest bill was in January

```

Since the program uses two arrays larger than 11 elements, a **DIM** statement is required to define the array's sizes. In this case, both arrays are dimensioned to 13 elements (0 through 12), although element 0 is never used. After dimensioning the arrays, the program uses a loop to read the months of the year into the MO\$ array, and to input the monthly charges into the BILL array.

The program sequence beginning with line 170 is responsible for finding and displaying the yearly total, the average, the month of

lowest charge, and the month of the highest charge. The sequence begins by setting TOTAL equal to 0, and setting SMALLEST and LARGEST to 1. The variables SMALLEST and LARGEST will be used to store the *number of the array element* with the smallest and largest values.

The program then enters the loop which computes the total and finds the two charges. The total is found by adding the 12 array elements. The method used to find the smallest and largest charges is described below.

1. When the loop begins, it is assumed that array element 1 contains the smallest and largest value. This provides a starting point for the first comparison.
2. On each execution of the calculation loop, the program compares the value of the current array element with the value of the array element it has previously found to be the smallest. If the current array element is found to be smaller, the value of SMALLEST is set equal to the number of the current array element. On the next execution of the loop, the comparison will be repeated with the new smallest element.
3. An identical technique is used to find the largest array element.
4. When the calculation loop is complete, SMALLEST will equal the number of the array element with the smallest charge, and LARGEST will equal the number of the element with the largest charge. (If there are two identical smallest or largest charges, the program will find only the first.)

The program then displays the total, average, and month of the smallest and largest charges. Since the elements of the two arrays have a one-to-one correspondence, MO\$(SMALLEST) will contain the name of the month with the smallest charge, and MO\$(LARGEST) will contain the name of the month with the largest charge.

The preceding examples have illustrated what are known as *one-dimensional* arrays. A one-dimensional array is essentially a numbered list, where the number of the item in the list corresponds to the number of the array element. It is also possible to have *two-*

	column 0	column 1	column 2	column 3
row 0	5	9	1	3
row 1	7	4	3	0
row 2	2	8	6	5

Fig. 19-1 An example of a table of values illustrating how each element of a two-dimensional array is defined by a row value and a column value.

(and *three-, four-, and larger*) *dimensional arrays*. (The number of dimensions permitted depends only on the amount of memory space available.)

A two-dimensional array can be visualized as a table of values, as illustrated in Fig. 19-1. Any element in a two-dimensional array can be identified by a row number and a column number. For example, the value in row 2, column 1, is 8.

As with one-dimensional arrays, the size of a two-dimensional array is defined with a **DIM** statement. The general form of a two-dimensional **DIM** statement is

DIM *array name* (*row size*,*column size*)

where *row size* is the number of the largest row and *column size* is the number of the largest column. The row value is always specified before the column value. You are not required to use a **DIM** statement for an array that has 11 or fewer rows and 11 or fewer columns.

A **DIM** statement to define an array suitable for storing the table of values given in Fig. 19-1 is shown below.

DIM TABLE(2,3)

Notice that this statement defines an array that is three rows by four columns in size.

A program that allows you to enter values into an array that is four rows by two columns in size is shown below. The program then allows you to display the value of any element in the array.


```

NEW
100 ' 2 Dimensional example
110 DIM TABLE(3,1)
120 FOR ROW=0 TO 3
130 INPUT "Sample number";TABLE(ROW,0)
140 INPUT "Sample number";TABLE(ROW,1)
150 NEXT ROW
160 INPUT "Enter row value";ROW
170 INPUT "Enter column value";COLUMN
180 PRINT "Element";ROW;",";COLUMN;"=";TABLE
    (ROW,COLUMN)
190 GOTO 160

```

A sample **RUN** is given below.

```

RUN
Sample number? 111
Sample number? 888
Sample number? 454
Sample number? 333
Sample number? 222
Sample number? 999
Sample number? 101
Sample number? 717
Enter row value? 2
Enter column value? 0
Element 2 , 0 = 222
Enter row value? 0
Enter column value? 0
Element 0 , 0 = 111
Enter row value?

```

(Press **CTRL** **C** to stop program execution.)

Since the TABLE array has been defined as having only four rows and two columns, a ?BS Error message will be displayed if you enter a row value larger than 3 or a column value larger than 1. Entering a negative value will also cause the error to occur.

SUMMARY

Array variables have the capability to store many different values under one variable name. The different values assigned to an array variable are called *elements*. Each element is identified by a unique *subscript*. The first element of any array is element 0.

The rules for forming array names are identical to those for forming simple variable names with the added restriction that the array element be specified by a subscript enclosed in parentheses immediately following the array name. The type declaration characters and global declaration statements apply to array variables just as they do to simple variables. For example, if an array is to be restricted to single precision, you must declare the array to be a single-precision variable using the ! declaration character or the **DEFSNG** statement. (As with simple variables, the default variable type for undeclared array variables is double precision.)

The **DIM** statement is used to define the size of an array. The general form of the statement for a one-dimensional array is

DIM *array name* (*array size*)

where *array name* is the name of the array variable and *array size* is the number of the largest element that will be used. Since the first element in an array is element 0, the total number of elements defined by a **DIM** statement is given by the value of *array size* + 1. A single **DIM** statement can dimension more than one array if the different array names are separated by commas.

The following rules must be followed when dimensioning arrays:

1. The **DIM** statement must always be executed before the first use of an array variable by a program.
2. An array can be dimensioned only once during the execution of a program. Thus, the size of an array cannot be changed by a program after it has been defined. If you attempt to redimension an array, a ?DD Error (Doubly Dimensioned array) message is displayed.

You are not required to have a **DIM** statement for one-dimensional arrays with 11 or fewer elements. Nevertheless, it is a good idea

to dimension even these small arrays, since dimensioning for fewer than 11 elements saves memory over not dimensioning at all.

Two-dimensional and larger-dimensional arrays can also be created by the **DIM** statement. The general form of a two-dimensional **DIM** statement is

DIM *array name*(*row size*,*column size*)

where *row size* is the number of the largest row and *column size* is the number of the largest column. The row value is always specified before the column value. As with one-dimensional arrays, two-dimensional arrays of 11 or fewer rows and 11 or fewer columns do not have to be dimensioned, although defining the smaller arrays can save memory.

The GOSUB and RETURN Statements

As you develop programs that are longer and more complicated, you will find that it is often necessary to repeat the same series of actions at different locations in a program. The most obvious way to write such a program is to reenter the necessary program lines each time they are needed. Of course, this method of handling repetition increases the size of the program and makes it more difficult and time consuming to write.

Fortunately, BASIC provides an important alternative known as a subroutine. A subroutine is a program segment that can be executed whenever needed by means of a **GOSUB** statement. The **GOSUB** statement has the general form

GOSUB *line number*

and instructs the computer to save a *return address* and then transfer program execution to the line number specified by the **GOSUB**

statement. The return address consists of the location of the first program statement following the **GOSUB** statement.

Every subroutine must end with a **RETURN** statement. When the **RETURN** statement of a subroutine is executed, the computer sends program execution back to the address that was saved when the **GOSUB** statement was executed.

A program that illustrates the operation of the **GOSUB** and **RETURN** statements by displaying a message indicating what part of the program is being executed is shown below.

NEW	
100 ' Subroutine demonstration program	
110 PRINT "Executing main program"	
120 GOSUB 170	
130 PRINT "Executing main program again"	
140 GOSUB 170	
150 PRINT "Main program yet again"	
160 END	
170 ' Beginning of the subroutine	
180 PRINT "Now executing the subroutine"	
190 RETURN	

] Main program
] Subroutine

A sample **RUN** of the program follows.

```

RUN
Executing main program
Now executing the subroutine
Executing main program again
Now executing the subroutine
Main program yet again
Ok
  
```

When the computer executes the **GOSUB** 170 statement, it transfers execution to line 170, just as if a **GOTO** statement were executed. Since the **GOSUB** statement saves a return address, however, the **RETURN** statement in the subroutine allows execution to resume with the appropriate main program line number. If a **GOTO** statement had been used instead of the **GOSUB** statement, no return address

would be stored and the program would not know to which line number to return. Thus, the power of a subroutine is that it can be executed at any place in the main program without permanently changing the order of program execution.

To emphasize that a **GOSUB** statement transfers execution to another location only temporarily, programmers generally refer to the process of executing a subroutine as "calling a subroutine." The word "call" is understood by programmers to indicate that execution will return to the main program when the subroutine is complete.

Notice the **END** statement in line 160 of this program. This statement serves a very important purpose. If you remove the **END** statement, the computer will allow execution to continue into the subroutine after line 150 is executed. The first line of the subroutine will execute properly since it is a **PRINT** statement, but when the computer attempts to execute the **RETURN** statement, a ?RG Error (Return with Gosub) message will be displayed. It is always a mistake to allow a subroutine to be executed by a method other than a **GOSUB** statement, since no return address is created for the **RETURN** statement to use.

The computer uses a "last address in-first address out" method of keeping track of return addresses. Each time a **GOSUB** statement is executed, a return address is stored. Each time a **RETURN** statement is executed, the return address stored *last* is erased from memory and used as the transfer address. This arrangement makes it possible for one subroutine to call another subroutine, a technique known as "nesting" subroutines.

To illustrate nesting subroutines, add the following lines to the subroutine demonstration program.

```
185 GOSUB 200
200 ' Beginning of second subroutine
210 PRINT "Executing subroutine 2"
220 RETURN
```

RUN the program.

```

RUN
Executing main program
Now executing the subroutine
Executing subroutine 2
Executing main program again
Now executing the subroutine
Executing subroutine 2
Main program yet again
Ok

```

Now, when the main program calls the subroutine at line 170, that subroutine displays its message and in turn calls a second subroutine at line 200. The second subroutine displays its message and then returns execution to line 190 of the first subroutine, which sends execution back to the appropriate line in the main program. Thus, program execution returns in the opposite order in which the subroutines are called.

It is also possible for the main program to call subroutine 2 directly. Make the following change to line 120 to demonstrate this feature.

```
120 GOSUB 200
```

A sample **RUN** appears below

```

Run
Executing main program
Executing subroutine 2
Executing main program again
Now executing the subroutine
Executing subroutine 2
Main program yet again
Ok

```

The next program provides a more practical example of the use of subroutines. Before entering this program, however, be certain that you understand how the **GOSUB** and **RETURN** statements affect the order of program execution. If you are unsure about some aspect of their operation, experiment with the subroutine demon-

stration program by adding subroutines and subroutine calls of your own. For example, you might want to add a third subroutine and have it called by subroutine 2. By writing a subroutine that displays a message identifying itself, the flow of the program is easy to follow.

The program that follows calculates the depreciation of an asset using the Accelerated Cost Recovery System (ACRS) adopted by the tax law of 1981. An asset life of three years is assumed. The function of the subroutine in the program is to calculate the amount of depreciation for a year and then round that value to two decimal places using the rounding formula discussed in Lesson 13. If a subroutine had not been used, it would be necessary to duplicate this calculation in three places in the program.

NEW	
100 ' Program to calculate ACRS depreciation	
110 INPUT "Enter cost of asset";ASSET	
120 F=.25	
130 GOSUB 220	
140 PRINT "Depreciation year 1 =";DEPR	
150 F=.38	
160 GOSUB 220	
170 PRINT "Depreciation year 2 =";DEPR	
180 F=.37	
190 GOSUB 220	
200 PRINT "Depreciation year 3 ="; DEPR	
210 END	
220 ' Depreciation subroutine	
230 DEPR=ASSET*F	
240 DEPR=INT(DEPR*100+.5)/100	
250 RETURN	

Main program

Subroutine

A sample **RUN** of the program is shown below.

```

RUN
Enter cost of asset? 2995.95
Depreciation year 1 = 748.99
Depreciation year 2 = 1138.46
Depreciation year 3 = 1108.5
Ok

```


In addition to providing another example of the operation of the **GOSUB** and **RETURN** statements, the ACRS program also illustrates how information can be passed between the main program and a subroutine. Just before calling the subroutine, the main program assigns the value to F that the subroutine is to use in its calculations. The subroutine assigns the result of its calculations to the variable DEPR, which is then displayed by the main program. Thus, the fact that the values of variables are not changed by the flow of program execution is used to send and get information from the subroutine.

Here is a much longer subroutine example.

```

NEW
100 'Cards program
110 DIM CARD$(52)
120 DEFINT B—Z 'To speed up program execution
130 A=RND(—VAL(RIGHT$(TIME$,2))—1)
140 GOSUB 350 'Read cards into array
150 PRINT 'To cancel pending display state
160 PRINT "Shuffling deck . . .";
170 GOSUB 200 'Shuffle deck
180 GOSUB 290 'Deal cards
190 GOTO 150 'Repeat loop
200 ' Shuffle deck subroutine
210 FOR COUNT=1 to 50
220 X=INT(52*RND(1)+1)
230 Y=INT(52*RND(1)+1)
240 TEMP$=CARD$(X)
250 CARD$(X)=CARD$(Y)
260 CARD$(Y)=TEMP$
270 NEXT COUNT
280 RETURN
290 ' Deal cards subroutine
300 S=INT(47*RND(1)+1)
310 FOR COUNT=S TO S+4
320 PRINT CARD$(COUNT); " ";
330 NEXT COUNT
340 RETURN

```

```

350 ' Read cards subroutine
360 FOR COUNT=1 TO 52
370 READ CARD$(COUNT)
380 NEXT COUNT
390 RETURN
400 DATA 2S,3S,4S,5S,6S,7S,8S,9S,10S,JS,QS,KS,AS
410 DATA 2H,3H,4H,5H,6H,7H,8H,9H,10H,JH,QH,KH,AH
420 DATA 2D,3D,4D,5D,6D,7D,8D,9D,10D,JD,QD,KD,AD
430 DATA 2C,3C,4C,5C,6C,7C,8C,9C,10C,JC,QC,KC,AC

```

The function of this program is to create a deck of cards, shuffle the cards, and then deal out a hand of five cards. A sample **RUN** of the program is given below. Since the randomizing expression (discussed in Lesson 15) is used, the cards that you see in your display and those shown in this text will differ.

```

RUN
Shuffling deck . . . 10C JH 8H 6S AH
Shuffling deck . . . 8S 10S 10C 3D 3H
Shuffling deck . . . 4S AH 2D 3C 5H

```

(Press **CTRL** **C** to stop program execution.)

The abbreviations used to represent the suits are S for spades, H for hearts, D for diamonds, and C for clubs. The abbreviations for the nonnumeric card values are J for jack, Q for queen, K for king, and A for ace.

The first point to observe about the card program is that the subroutines are not used to perform a function needed several times by the main program. Then why use them? The answer is that the subroutines split the program into units that are easier to understand. In fact, many programmers consider the organizational value of subroutines to be as important as the capability to reduce repetition. By separating the overall task of a program into smaller tasks and writing subroutines to perform those pieces, you can reduce complicated programs to manageable portions.

Briefly, here is how the card program works. The program begins by defining an array named **CARD\$** to hold the 52 cards. Next,

the randomizing expression is used to ensure a different seed number each time the program is executed. A subroutine is then called to read the card values from the **DATA** statements into the **CARD\$** array. This needs to be done only once each time the program is executed.

The subroutine beginning at line 200 is used to shuffle the deck of cards. The technique adopted for this process is to select two cards at random and then exchange their positions in the **CARD\$** array. The cards to be exchanged are selected by randomly assigning values between 1 and 52 to the variables **X** and **Y**. Then, while the original value of **CARD\$(X)** is temporarily stored in the variable **TEMP\$**, the value of **CARD\$(Y)** is put into **CARD\$(X)**. To complete the exchange, the value of **TEMP\$** is assigned to **CARD\$(X)**. To mix the deck thoroughly, it is necessary to repeat this process a number of times. The **FOR NEXT** loop in the subroutine repeats this sequence 50 times.

After the cards are shuffled, the subroutine beginning at line 290 is called to deal the hand. This subroutine begins by generating a random starting value between 1 and 48 so that the dealing begins at a different location each time (this is similar to "cutting" the deck). The upper value of this variable is placed at 48 so that the program will never attempt to deal past **CARD\$(52)**. The **FOR NEXT** loop that follows displays five cards beginning with the randomly selected starting card. A semicolon is placed after the **PRINT** statement to create a pending display state so that the cards can be displayed on one line.

When execution returns from the dealing subroutine, the **GOTO** statement in line 190 loops the program back to the shuffling operation.

SUMMARY

Subroutines are executed by means of the **GOSUB** statement. The **GOSUB** statement instructs the computer to save a return address and then transfer execution to the line number following the **GOSUB** statement. Every subroutine must end with a **RETURN** statement. The purpose of the **RETURN** statement is to transfer program execution back to the return address created by the **GOSUB** statement.

To indicate that executing a **GOSUB** statement sends execution to another location only temporarily, programmers generally refer to the process of executing a subroutine as “calling a subroutine.”

The *last address in/first address out* method of keeping track of subroutine addresses allows subroutines to call other subroutines, a technique known as “nesting” subroutines. When subroutines are nested, the flow of execution always returns in the opposite order in which the subroutines were called.

Besides reducing the occurrence of repetition in a program, subroutines are an organizational tool. They allow large programming applications to be broken into smaller tasks that are more easily written and understood.

REVIEW TEST 4

1. Which of the following are valid BASIC statements?

- (a) 135 FOR F=T+7 TO A*L
- (b) 200 DATA 126;17;5;15;7
- (c) 212 FOR F=-10 TO 20
- (d) 116 READ A,A,A,A
- (e) 999 DATA 1,6,3/2,16
- (f) 180 READ A+2,A+3,A+4
- (g) 110 FOR K=0 TO 0
- (h) 101 DIM A(5.2)
- (i) 712 RESTORE 100,200
- (j) 100 DIM 3D(17)
- (k) 211 WD(3)\$="test"
- (l) 500 DIM SWAT(0)
- (m) 300 FOR A(2)=1 TO 10
- (n) 500 FOR A=A(1) TO 10
- (o) 222 DIM A(G+7)
- (p) 100 DIM WEEK\$(7),ID(12,7)
- (q) 950 GRADE (2)=GRADE2+GRADE2!
- (r) 951 DIM PRIME(32),PRICE(3,7)
- (s) 200 GOSUB 66112

2. Write a program that uses a **FOR NEXT** loop to sum the whole numbers between 1 and 200, inclusive ($1 + 2 + 3 + \dots + 200$).
3. A very rich man agrees to the following terms: for 31 days he will double the amount of money that he paid on the previous day. He begins by paying \$.01 on day 1. Write a program that displays the following information for every day in the 31-day period: the number of the day (beginning with 1 for the first day), the amount to be paid that day, and the total wages paid (including the amount to be paid that day).
4. What are the two methods of placing comments in a program? How do they differ?
5. What is the purpose of dummy values in **DATA** statements?
6. What is the purpose of the **RESTORE** statement?
7. What is an array element? Does the variable PAIR\$(5,6) describe one or two array elements?
8. What is the purpose of a **DIM** statement? When is it required?
9. To dimension the A array for 20 elements, would you use A(19) or A(20)?
10. How many array elements are defined by the following statement?

130 DIM TEST(3,7,4)
11. What is the function of the **RETURN** statement?

12. How are subroutines helpful in improving the organization of a program?
13. When is an **END** statement required in a program?
14. How is information passed to a subroutine and back to the main program?

Using PRINT USING

Lesson 1 introduced the **PRINT** statement and illustrated its use. In addition to the colon and semicolon *expression list* options discussed in that lesson, the **PRINT** statement allows the use of an optional **USING** parameter.

The **USING** parameter instructs the computer to follow a specific format when displaying information. This format is defined by a special *format string* that follows the **USING** parameter and is separated from the *expression list* by a semicolon. The general form of the **PRINT USING** statement is

PRINT USING *format string*;*expression list*

The *format string* can be either a string constant (enclosed in quotes) or a string variable.

The characters available to define the *format string* are listed below.

For String Data

- ! Instructs the computer to print only the first character of a string.
- \spaces\ Instructs the computer to print a specific number of characters from a string. The number of characters printed is 2 plus the number of spaces enclosed within the backslashes (\). If no spaces are enclosed within the backslashes, two characters are printed (the minimum). The reason for the two-character-minimum rule is that the format characters themselves count as part of the format. Thus, the two backslashes specify two characters, whether or not they enclose any spaces. (The \ character is entered using the **GRAPH**  key combination on the Models 200 and 100.)

For Numeric Data

- # Specifies the position of a digit. The amount of # signs used in the format establishes the maximum number of digits that can be displayed. If a number is larger than the *format string*, a percent (%) sign is displayed to the left of the number to indicate a formatting error.
 - + Instructs the computer to print the sign of the number. If the + character is placed at the beginning of the *format string*, the sign is printed preceding the number. If the + is placed at the end of the *format string*, the sign is printed following the number.
 - Instructs the computer to print the sign of a negative number. The sign can be displayed preceding or following the number, depending on the location of the - sign in the *format string*.
 - .
- Specifies the position of the decimal point in a number format. If omitted, no digits are displayed to the right of the decimal point. If the decimal

point is used, digits to the right of the decimal point are rounded to the number of places specified by the *format string*.

,

Instructs the computer to insert commas in the printed number. The commas are inserted every three digits to the left of the decimal point. A single comma instructs the computer to insert as many commas as necessary, depending on the size (magnitude) of the number. The comma must be left of any decimal point in the *format string*, but cannot be the leftmost character in the *format string*.

\$\$

Instructs the computer to print a dollar (\$) sign preceding the number. The \$\$ format characters, if used, must be the leftmost characters in the *format string*.

**

Instructs the computer to change any leading blanks in a number to asterisks. The asterisks must be to the left of any # signs in the *format string*.

**\$

Instructs the computer to fill leading blanks with asterisks and print a dollar sign preceding the number.

^^^

Instructs the computer to print a number in exponential (scientific) notation. For a discussion of scientific notation, refer to the manuals supplied with your computer. [The caret (^) is entered using the **SHIFT** 6 key combination.]

An example illustrating the **PRINT USING** statement is shown below.

```
NEW
100 'Example of PRINT USING statement
110 INPUT "Enter a sample number";N
120 PRINT USING " # # # # # #.# #";N
130 GOTO 110
```

A sample **RUN** appears below.

```

RUN
Enter a sample number? -23
      -23.00
Enter a sample number? .3456
      0.35
Enter a sample number? 362432
    362432.00

```

(Press **CTRL** **C** to halt program execution.)

Observe that the computer fits the numbers into the format given by the *format string*. Since the *format string* specifies two digits to the right of the decimal point, the computer either displays zeros when there are no decimal digits in the number being printed, or rounds to two decimal places when the number has more than two decimal digits. Notice also that while unused format positions to the left of the decimal point are displaced as spaces, the "sign" space that is ordinarily displayed in front of every positive number is omitted.

The six # signs to the left of the decimal point limit the number of digits that can be displayed there. If you enter a number with more than six digits to the left of the decimal, the computer displays the number, but places a % sign to the left of the number to indicate that a formatting error has occurred.

```

RUN
Enter a sample number? 1234567
%1234567.00
Enter a sample number? -123456
%-123456.00

```

(Press **CTRL** **C** to halt program execution.)

The minus sign is counted as a character, so -123456 is also too large to fit the *format string*.

A *format string* can contain information other than format characters. To illustrate this, replace line 120 with

```
120 PRINT USING "The number is: # # # # # #.# #";N
```

and **RUN** the program again.

```
RUN
Enter a sample number? 59.999
The number is:      60.00
```

(Press **CTRL** **C** to halt program execution.)

Nonformat characters that are used in a *format string* are referred to as *literals*, because they are displayed exactly as you enter them. You can use any character that you want as a literal except for a quotation mark.

The **USING** parameter will also format string information. To illustrate this possibility, make the following changes to the demonstration program (line 120 has seven spaces between the backslashes).

```
110 INPUT "Enter a sample string";S$
120 PRINT USING "The string is: \      \";S$
```

A sample **RUN** is shown below.

```
RUN
Enter a sample string? Dallas
The string is: Dallas
Enter a sample string? Albuquerque
The string is: Albuquerque
```

(Press **CTRL** **C** to halt program execution.)

The number of spaces between the backslashes plus two determines the number of characters that are displayed.

A *format string* can specify a format for more than one variable at a time. The following program illustrates this feature. (Line 110 has six spaces between the backslash pairs.)

```
NEW
100 'Grocery list program
110 F$="Buy # \      \ \      \"
```

```

120 READ DESC$,QUANTITY,UNIT$
130 IF QUANTITY=-1 THEN END
140 PRINT USING F$;QUANTITY;UNIT$;DESC$
150 GOTO 110
160 DATA eggs,2,dozen,milk,1,quart
170 DATA buns,3,packages,lettuce,2,heads
180 DATA drinks,7,liters
999 DATA xxx,-1,xxx 'Dummy Values

```

A sample **RUN** is shown below.

```

RUN
Buy  2 dozen      eggs
Buy  1 quart      milk
Buy  3 packages   buns
Buy  2 heads      lettuce
Buy  7 liters     drinks
Ok

```

The three groups of format characters define the display format of the three variables.

SUMMARY

The **PRINT USING** statement instructs the computer to follow a specific format when displaying information. The general form of the **PRINT USING** statement is

PRINT USING *format string;expression list*

The *format string* can be either a string constant or a string variable. The characters available for defining a *format string* are summarized below.

Character	Function
!	Print first character of a string
\spaces\	Print <i>spaces</i> +2 string characters
#	Specifies the position of one digit
+	Print sign of number in position indicated by + sign

—	Print sign of negative number in position indicated by — sign
.	Specifies the position of the decimal point
,	Print a comma every three digits (left of the decimal point)
\$\$	Print a dollar sign preceding a number
**	Print asterisks in place of leading spaces
**\$	Print a dollar sign preceding a number and asterisks in place of leading spaces
^ ^ ^ ^	Print a number in exponential notation

Any characters in a *format string* other than those listed above are known as *literals* and are displayed “as is.” The amount of # signs used in the format establishes the maximum number of digits or characters that can be displayed.

Multiple Statement Lines

In the examples you have seen so far, each program statement begins with a new line number. It is possible, however, to put more than one program statement on a line. Multiple statements can be placed on the same line by entering a colon between the statements. Except when used with an **IF THEN** statement as explained later in this lesson, a colon instructs the computer to treat the statement that follows as if it were an entirely separate program line.

To illustrate this feature, look at the following program.

```
100 INPUT "FEET";FEET
110 METERS=FEET*.3048
120 PRINT "Meters =";METERS
```

The purpose of this program is to convert feet to meters. Here is the same program as one multistatement line.

```
100 INPUT "Feet";FEET:METERS=FEET*.3048:PRINT
    "Meters =";METERS
```

To combine the program lines, simply substitute a colon for the number of the next line. As long as the line does not exceed the 250-character limit imposed by BASIC, you can continue to add new statements to a line by preceding them with a colon. (The example given above is 61 characters long.)

As you compare the two versions of the English-to-metric conversion program, it is evident that the first is easier to understand. Then why ever use multistatement lines? The primary answer is to save memory space. The computer uses 4 bytes of memory to store a line number, but only 1 byte to store a colon separator. Thus, replacing two line numbers with colons reduces the amount of memory needed to store the English-to-metric program by 6 bytes. This is calculated as follows:

memory for 2 line numbers	=	2 × 4	=	8 bytes
memory for 2 colons	=	2 × 1	=	<u>2 bytes</u>
savings	=			6 bytes

When memory space is not critical, however, it is usually better to avoid using multistatement lines, since they increase the complexity of a program. It is not uncommon for programmers to introduce errors into a program as a result of trying to save memory by combining lines. The more densely packed lines seem to breed errors.

Also, be aware that some program lines cannot be correctly combined. For example, a program such as

```
100 X=100
110 PRINT X
120 X=X+2
120 GOTO 110
```

cannot be combined into one line because the **GOTO** statement can only transfer execution to the beginning of a line—not to the middle as would be required if this program were entered as one

multistatement line. You *can* reduce this program to two lines, however.

```
100 X=100
110 PRINT X:X=X+2:GOTO 110
```

The computer generally treats what follows a colon statement separator as an entirely separate program line. The exception to this rule occurs when the colon separator is used with an **IF THEN** statement. To understand why the colon separator is treated differently in this case, you need to remember that the general form of the **IF THEN** statement is

IF condition THEN action

If the *condition* is true, the *action* is executed. If it is false, the *action* is skipped. Therefore, if a second statement is added to the action by means of a colon separator, that statement will also execute *only* when the condition is true; it will be skipped when the condition is false. The two examples that follow should help clarify this point.

Example 1

```
100 INPUT "Enter a number";N
110 IF N<0 THEN PRINT "Negative"
120 GOTO 100
```

Example 2

```
100 INPUT "Enter a number";N
110 IF N<0 THEN PRINT "Negative":GOTO 100
```

At first glance, these programs may appear to be functionally identical. They are not, although the only difference is the use of the colon separator. In Example 1, the computer is in a true endless loop. No matter what numeric value you enter, the computer will always send execution back to line 100.

In Example 2, the computer is not in an endless loop. The program will stop when you enter a positive value. The reason for this differ-

ence is that the **GOTO** statement in Example 2 is executed only when the condition is true—in this case, only when a negative number is entered. When a zero or positive value is entered, both the **PRINT** and **GOTO** statements are skipped.

The computer's treatment of the colon separator following an **IF THEN** statement is a useful feature. It simplifies the process of making multiple operations conditional. For example, consider the following program.

```

100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 N1=INT(9*RND(1)+1)
120 N2=INT(9*RND(1)+1)
130 PRINT N1;"*";N2;"=" ";
140 INPUT ANSWER
150 IF ANSWER=N1*N2 THEN 180
160 PRINT "WRONG. TRY AGAIN."
170 GOTO 130
180 PRINT "VERY GOOD!"
190 GOTO 110

```

This is a copy of the multiplication tables program discussed in Lesson 15. The purpose of the **IF THEN** statement in line 150 is to check the answer that is entered. If the answer is correct, the programmer wants to display VERY GOOD and send execution back to generate a new problem. Because the program does not use multistatement lines, it is necessary to transfer execution to another part of the program to perform these actions. A version of the program that uses a multistatement line is given below.

```

100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 N1=INT(9*RND(1)+1)
120 N2=INT(9*RND(1)+1)
130 PRINT N1;"*";N2;"=" ";
140 INPUT ANSWER
150 IF ANSWER=N1*N2 THEN PRINT "VERY
    GOOD!":GOTO 110
160 PRINT "WRONG. TRY AGAIN."
170 GOTO 130

```

The second version is not just shorter—it is also more understandable. The consequences of a true decision-making result are obvious. In the original version, by contrast, it is necessary to examine several parts of the program to find the consequences of a true result.

SUMMARY

Multiple statements can be placed on the same program line by entering a colon between the statements. Except when used with **IF THEN** statements, the colon instructs the computer to treat the statement following the colon as if it was an entirely separate program line. When a colon separator is used following an **IF THEN** statement, the additional statement or statements are part of the *action* of the **IF THEN** statement. This is a useful feature that simplifies the execution of multiple actions by **IF THEN** statements.

The maximum line length is 250 characters.

Combining program statements shortens programs and reduces the memory needed to store them. This reduction in size is usually gained at the expense of program clarity. Unless program size is crucial, it is generally better to avoid densely packed multistatement lines.

More on Decision Making

The purpose of the **IF THEN** statement in a program is to permit the computer to perform different actions depending on the result of a decision-making test. If the test is true, the action following the **THEN** portion of the **IF THEN** statement is executed. If the test is false, the action is skipped.

By using the **ELSE** option with the **IF THEN** statement, you can define a specific action to be performed when the result of the test is false. The general form of the **IF THEN** statement with the **ELSE** option is

IF condition THEN action1 ELSE action2

The **IF THEN ELSE** combination instructs the computer to perform *action1* when the *condition* is true and *action2* when the *condition* is false. In other words, "do *action1* if true, do *action2* if false."

To illustrate the convenience of the **ELSE** option, examine the following program.

```
100 INPUT "Enter a number";N
110 IF N<0 THEN PRINT "Negative"
120 PRINT "Positive"
130 GOTO 100
```

Here the programmer wants to indicate whether the entered number is negative or positive (zero is assumed to be positive). The logic used to perform this task is not correct, however. Although the program does perform as intended when a positive number is entered, it does not work correctly with negative numbers. When a negative number is entered, the program displays both messages. It executes the action following the **THEN** statement and the program line that follows.

To fix this program, you can replace line 110 with

```
110 IF N<0 THEN PRINT "Negative":GOTO 100
```

or you can use the **ELSE** option. The program as it appears if you use the **ELSE** option is shown below.

```
100 INPUT "Enter a number";N
110 IF N<0 THEN PRINT "Negative" ELSE PRINT "Positive"
120 GOTO 100
```

Although both the corrected original and the **ELSE** version perform the intended task, the **ELSE** version has fewer program lines and is easier to understand.

When multiple statements are used following an **ELSE** option, they are treated as part of the action to be performed when the test result is false. For example, in the sample program line

```
300 IF N=38 THEN PRINT "Done":END ELSE READ
    T:N=N+T:GOTO 100
```

action1 consists of

```
PRINT "Done":END
```

and *action2* of

READ T:N=N+T:GOTO 100

Notice that although colons are needed to separate multiple statements, no colon is needed to separate the **ELSE** option.

SUMMARY

The **ELSE** option is used to define a specific action to be performed when the result of a decision-making test is false. The general form of the **IF THEN** statement with the **ELSE** option is

IF *condition* THEN *action1* ELSE *action2*

Action1 and *action2* can consist of single statements or multiple statements joined by the colon separator.

The advantage of the **ELSE** option is that it shortens a program and generally makes it easier to understand.

More on FOR NEXT Looping

The purpose of **FOR TO NEXT** statements in a program is to control the number of times that a loop is executed. In the examples in which these statements have been used so far, only one **FOR NEXT** loop has been active at a time. It is possible, however, to “nest” **FOR NEXT** loops just as you can nest subroutines.

An example that demonstrates the operation of nested **FOR NEXT** loops is shown in Fig. 24-1. As you can see from the boxes drawn around the loops, the inner loop is completely within the outer loop. This is absolutely essential when nesting **FOR NEXT** loops. If you attempt to put any portion of the inner loop outside the outer loop, an error condition will result.

NEW

100 'Nested FOR NEXT demonstration

110 FOR O=1 to 3

120 PRINT "Outer loop, O =";O

130 FOR I = 1 to 2

140 PRINT " Inner loop, I =";I

150 NEXT I

160 NEXT O

—Inner loop

—Outer loop

Fig. 24-1 Program demonstrating the operation of nested **FOR NEXT** loops.

A sample **RUN** of the program from Fig. 24-1 is shown below.

```

RUN
Outer loop, O = 1
    Inner loop, I = 1
    Inner loop, I = 2
Outer loop, O = 2
    Inner loop, I = 1
    Inner loop, I = 2
Outer loop, O = 3
    Inner loop, I = 1
    Inner loop, I = 2
Ok
  
```

The program executes the entire inner loop every time it executes the outer loop once. Therefore, the inner *control variable* always begins anew with the *initial value*. While the inner loop is being executed, however, the computer keeps track of the current value of the *control variable* of the outer loop. When the value of this variable exceeds 3, the program stops.

You can use as many nested loops in a program as you desire. When nesting loops, however, observe the following rules.

1. Each loop must begin with its own **FOR TO** statement and end with its own **NEXT** statement.
2. An inside loop must be located completely within the bounds of the next higher loop.
3. Each loop must have a different *control variable*.

Nested loops are very useful when you want to find all possible combinations, or permutations, of two or more variables. For example, suppose that you want to display the multiplication table up to 6×6 . The program shown in Fig. 24-2 demonstrates how this is done with nested **FOR NEXT** loops.

NEW

100 'Multiplication table program

110 FOR N1=0 TO 6

120 FOR N2=0 TO 6

130 PRINT N1; " * "; N2; " = "; N1*N2

140 NEXT N2

150 NEXT N1

—Inner loop

—Outer loop

Fig. 24-2 Program demonstrating how nested **FOR NEXT** loops are used to find all permutations of the multiplication table up to 6×6 .

A partial **RUN** of the program from Fig. 24-2 is shown below.

RUN

0 * 0 = 0

0 * 1 = 0

0 * 2 = 0

.

.

.

6 * 5 = 30

6 * 6 = 36

Ok

The first number in each of the multiplication problems is generated by the outer loop. The second number is generated by the inner loop. The two loops together generate all possible combinations of the values between 0 and 6.

The power and flexibility of **FOR NEXT** loops can be greatly increased by using the **STEP** option. The **STEP** option allows you to define a specific increment to use for increasing the *control variable*. The general form of the **FOR NEXT** statement with the **STEP** option is

FOR *control variable* = *initial value* TO *limit* STEP *increment*

The value given for *increment* can be either a constant or a variable. The **STEP** option is useful primarily in cases where the *control variable* is being used for several purposes. For example, in the compound interest program developed in Lesson 16, the value of the *control variable* was also used as the annual interest rate. That program is reproduced below.

```
100 INPUT "Principal";P
110 FOR COUNT=10 TO 15
120 I=COUNT/100
130 FV=P*(1+I/12)^12
140 FV=INT(FV*100+.5)/100
150 PRINT "Value at";COUNT;"% =" ;FV;"dollars"
160 NEXT COUNT
```

By replacing line 110 of this program with

```
110 FOR COUNT=10 to 15 STEP .5
```

you can calculate the effects of increasing the interest rate by .5 percentage points. The **STEP .5** statement instructs the computer to increase the value of COUNT by .5 rather than by 1. A sample **RUN** of the new version of the program is shown below.

```
RUN
Principal? 987.23
Value at 10 %=1090.61 dollars
```

```

Value at 10.5 %=1096.03 dollars
Value at 11 %=1101.47 dollars
Value at 11.5 %=1106.94 dollars
Value at 12 %=1112.44 dollars
Value at 12.5 %=1117.96 dollars
Value at 13 %=1123.5 dollars
Value at 13.5 %=1129.07 dollars
Value at 14 %=1134.66 dollars
Value at 14.5 %=1140.29 dollars
Value at 15 %=1145.93 dollars
Ok

```

By changing the *increment* of the **STEP** option, you can change the value by which the interest rate is increased. For example, to observe the effects of increasing the interest rate by two percentage points, replace line 110 with

```
110 FOR COUNT=10 TO 15 STEP 2
```

and **RUN** the program again.

```

RUN
Principal? 987.23
Value at 10 % = 1090.61 dollars
Value at 12 % = 1112.44 dollars
Value at 14 % = 1134.66 dollars
Ok

```

Since the *control variable* is now being increased by two, the variable's *limit* is exceeded when 2 is added to 14.

Besides specifying a positive *increment*, the **STEP** option can specify a negative *increment*. If you then use the correct values for *initial value* and *limit*, a **FOR NEXT** loop can be made to count backward. To illustrate this feature, make the following change to line 110.

```
110 FOR COUNT=15 TO 10 STEP -1
```

A sample **RUN** is shown below.

```
RUN
Principal? 987.23
Value at 15 % = 1145.93 dollars
Value at 14 % = 1134.66 dollars
Value at 13 % = 1123.5 dollars
Value at 12 % = 1112.44 dollars
Value at 11 % = 1101.47 dollars
Value at 10 % = 1090.61 dollars
Ok
```

To make the **FOR NEXT** loop count backward, the *initial value* must be larger than the *limit*.

SUMMARY

FOR NEXT loops can be nested by placing one loop within another loop. When nesting **FOR NEXT** loops, the following rules must be observed.

1. Each loop must begin with its own **FOR TO** statement and end with its own **NEXT** statement.
2. An inner loop must be located completely within the bounds of an outer loop.
3. Each loop must use a different *control variable*.

The power and flexibility of **FOR NEXT** loops can be increased by using the **STEP** option. The **STEP** option permits a **FOR NEXT** loop to be incremented by any value, including negative or fractional values. The general form of the **FOR NEXT** statement with the **STEP** option is

FOR *control variable* = *initial value* TO *limit* STEP *increment*

The value given for the *increment* can be either a constant or a variable.

If a negative *increment* is used, and the *initial value* is larger than the *limit*, the **FOR NEXT** loop will count backward.

Storing Programs on Cassette Tape

Your computer is equipped with a built-in cassette interface that permits programs and data to be recorded on cassette tape for later retrieval and use. To use the cassette interface, you must have a compatible cassette recorder and a set of cables to connect the computer to the recorder (appropriate cables are available from the manufacturer of your computer).

The advantages of the cassette interface are at least threefold:

1. Cassette tape provides a method of "backing up" programs and data. Because the contents of the computer's RAM must be maintained by electrical power derived from the batteries installed in the computer, dead or weak batteries will cause the contents of memory to be lost. Memory information can also be lost if the computer malfunctions. The process of making duplicate copies of important programs

or data on storage media such as cassette tape is known as "backing up."

2. Cassette storage permits you to manage better the computer's most precious commodity—its random access memory (RAM). No matter what the memory size of your computer, at some point you will find that you have filled it to capacity. By recording programs and data that you are not actively using on cassette tape, and then deleting that information from the computer's memory, you can free memory for more current applications.
3. Cassette tapes provide a method of exchanging computer information. For example, almost all commercial suppliers of software for your computer distribute their product on cassette tape.

The cassette interface is designed for use with cassette recorders equipped with remote motor control. (A cassette recorder with remote motor control will have at least three jacks. The remote motor control jack is usually marked REM.) Remote motor control allows the computer to turn the cassette recorder motor on and off at appropriate times during the execution of cassette commands. Remote motor control also gives the computer the capability to read or record data files under program control (see Lesson 39 for more details on this aspect of cassette control).

For computer control of the cassette recorder to work properly, the computer must keep the recorder motor turned off except when actually reading or recording. Keeping the motor turned off allows you to press the recorder's PLAY or RECORD buttons before executing a cassette command and not have the recorder start until instructed to by the computer. On some brands of cassette recorders, you cannot rewind or fast-forward while remote control is turned off. If you have this type of recorder, you must unplug the remote motor control cable when you want to change the tape position.

The computer provides two commands for saving a BASIC program on cassette tape:

```
CSAVE "filename"  
SAVE "CAS:filename"
```

where *filename* represents the name to be assigned to the BASIC program. Filenames must begin with a letter of the alphabet and can be up to six characters in length. If the filename is a constant, it must be enclosed in quotes (you can use a string variable to identify the filename if you prefer).

Both commands perform the same operation—they save a copy of the program currently in the BASIC workspace to cassette tape. The **CSAVE** (Cassette **SAVE**) command is the traditional BASIC command for saving a program to cassette tape. The **SAVE** command can be used if you precede the *filename* with CAS: (the device name for the cassette interface).

Before executing either command, position the cassette tape where you want the recording to begin and press the RECORD button of the recorder.

Because of the importance of making backups, most computer users save more than one copy of a program. As an additional safeguard, you can verify that a program has been saved correctly with the

CLOAD? "*filename*"

command. Verification is an important safeguard against erroneous or bad recording of programs and is recommended as a routine practice. If a file does not verify as error-free, record the file again before you alter or clear the original program from the computer.

Before verifying a program, rewind the cassette tape to the beginning of the recording and press the PLAY button on the recorder. Then execute the **CLOAD?** command. The computer compares the program on the cassette tape with the program in the BASIC workspace. If any discrepancy is found, the message *Verify failed* is displayed.

To load a program from cassette tape into the computer's memory, use

CLOAD "*filename*"
LOAD "CAS:*filename*"

The volume setting of the cassette recorder is critical for proper loading of cassette files. Too high or too low a volume setting will

cause errors when reading the file. You may have to experiment with various volume settings before you find the correct setting for your cassette recorder.

Loading a program from cassette tape erases any program currently in the BASIC workspace. Therefore, when you have a program in the BASIC workspace that you want to preserve, save it as a RAM file before loading the cassette file.

On the Models 200 and 100, you can listen to the loading of the file through the computer's internal speaker by executing the

SOUND ON

command before loading the file. To turn the internal speaker off, use the

SOUND OFF

command. Some Models 200 and 100 computer owners have found that cassette files load more reliably if the internal speaker is turned off. (The PC-8201A does not have an internal speaker and consequently does not have a **SOUND ON/OFF** command.)

HINTS FOR USING THE CASSETTE INTERFACE

- When recording a file, be certain that the cassette tape is positioned past the leader and that there is sufficient tape remaining to contain the file. The leader is the nonmagnetic segment at each end of the tape.
- If your cassette recorder has a tape counter, use it to identify the starting and ending locations of your files. Label each tape so that you know the names of the files on the tape and the counter locations.
- Use unique and descriptive filenames.
- If you have forgotten the names of the files on a tape, you can make the computer list the filenames by **CLOADing** a filename that you know is *not* present on the tape. While

the **CLOAD** command is searching for the filename you specified, it will display the names of any other files it finds on the tape.

- Use shorter cassette tapes. Cassette tapes longer than 60 minutes are not recommended since the tape is thinner and is less reliably handled by most cassette recorders. As a rule, the effort spent to squeeze as many files as possible onto a longer tape does not justify the savings.
- If a tape's contents are of no value and you wish to use the tape for file storage, use a bulk eraser to clear the tape.
- Keep several backups of important files, on different tapes.
- Rewind each tape when you are through with it.
- Do not touch the tape surface. Avoid exposing tapes to contaminants such as cigarette smoke, ashes, dust, or moisture. Keep the tape out of direct sunlight and away from heat.

REVIEW TEST 5

1. Which of the following are valid BASIC statements?

- (a) 222 PRINT USING FMT\$,PRICE
- (b) 100 LOAD "CAS;" + FF\$
- (c) 250 PRINT USING S\$+T\$;H\$
- (d) 400 IF A=100 THEN 130 ELSE IF A=200 THEN 170
- (e) 100 PRINT USING "!! / /";"Joe";
 "Bob";"Green"
- (f) 200 IF FEE=1000 THEN FEE=FEE+15 GOTO 245
- (g) 110 GOSUB 200:REM CALL SUBROUTINE:X+X+1
- (h) 770 FOR X=1 TO 10:PRINT X;:NEXT X
- (i) 100 PRINT USING + ###.###;UNIT
- (j) 125 CSAVE "CAS:PGM1"
- (k) 135 SOUND OFF:READ A,B,C
- (l) 450 FOR COUNT=7 TO 1 ELSE -2
- (m) 500 PRINT USING "Grand Total=+**\$#,
 #####.###";SUM#
- (n) 300 FOR F=50 TO -50 STEP -5
- (o) 235 FOR A=B TO C STEP D
- (p) 266 FOR N1=19 TO 9 STEP N:T=R/T:NEXT N
- (q) 100 CLOAD? "PAYROLL"

2. How is a display format specified for the **USING** parameter?
3. How are multistatement lines created? When are they advantageous?
4. What is the function of the **ELSE** option in an **IF THEN** statement?
5. Write a program that uses the formula shown below to calculate and display the value of Y as X increases from 5 to 15 in increments of .5 unit.

$$Y = 17X^2 + X - 2$$

6. A common application of nested **FOR NEXT** loops is to read data into a two-dimensional array. Write a program that reads the following test scores from **DATA** statements into an array named **GRADE** and then calculates and displays the average test score for each student.

	Test 1	Test 2	Test 3	Average
Student 1	78	82	89	?
Student 2	83	85	81	?
Student 3	100	70	73	?
Student 4	65	68	71	?
Student 5	99	89	93	?
Student 6	87	0	93	?

7. What is the purpose of the following BASIC commands?
 - (a) SOUND OFF
 - (b) CSAVE "CARDS"
 - (c) CLOAD? "CARDS"
 - (d) LOAD "CAS:MENU"

Adding SOUND to a Program

You can enliven and enhance the effectiveness of your programs by adding sound effects to the programs. BASIC provides two instructions for generating sound effects: **BEEP** and **SOUND**. The **BEEP** statement produces a single half-second tone each time it is executed:

BEEP

The **SOUND** statement is more complex, permitting you to create a wide range of audio tones, including musical tones. The **SOUND** statement has the general form

SOUND *pitch,length*

The *pitch* parameter can range from 0 to 16383; the larger the *pitch* value, the deeper the tone. The *length* parameter can range

from 0 to 250; each increment of 50 in the *length* parameter represents approximately one second of duration. For example,

SOUND 9394,250

produces a tone that corresponds approximately to middle C for 5 seconds ($250 \div 50 = 5$). Figure 26-1 shows the *pitch* values that correspond to the usable musical tones produced by the computer. Use of other *pitch* values will produce nonmusical tones.

Note (Sharp/Flat)	Octave			
	1	2	3	4
G	12539	6269	3134	1567
g/a	11836	5918	2959	1479
A	11172	5586	2793	1396
a/b	10544	5272	2636	1318
B	9952	4976	2484	1244
C	9394	4697	2348	1174
c/d	8866	4433	2216	1108
D	8368	4184	2092	1046
d/e	7900	3950	1975	987
E	7456	3728	1864	932
F	7032	3516	1758	879
f/g	6642	3321	1660	830

Fig. 26-1 Table of *pitch* values for the **SOUND** statement corresponding to musical tones. An additional octave can be calculated by dividing the values of octave 4 by 2, but the tones are too high pitched to be useful.

Technical Note: The **SOUND ON** and **SOUND OFF** instructions, used to turn the internal speaker on and off when loading cassette files, does not affect the **BEEP** and **SOUND** *pitch,length* statements.

Tones are very useful in programming. They can, for example, indicate that a program or procedure is finished, warn that an error

(perhaps an entry error) has occurred, or simply add a lively dimension to a program.

If only one or two tones are being produced, the appropriate **SOUND** statements are generally just added to the program, perhaps in a subroutine if they are used several times. But if musical notes are being created, it is more common to store the *pitch* values in **DATA** statements. This provides a memory efficient method of storing the numbers.

The following example demonstrates this technique.

```
NEW
100 'Musical scales example
110 FOR S=1 TO 8
120 READ N:SOUND N,50
130 NEXT S
140 DATA 9394,8368,7456,7032
150 DATA 6269,5586,4976,4697
```

When you **RUN** this program, the computer will play the musical scales in the key of C (do, re, mi, . . . , do).

If it is necessary to be able to play the notes in different orders, simply reading them from **DATA** statements may not be satisfactory. It may be necessary to read the notes into an array for multiple accesses.

The following program ascends, and then descends, the musical scales.

```
NEW
100 'Musical scales example 2
110 DEFINT N,S:DIM N(8)
120 FOR S=1 TO 8
130 READ N(S):SOUND N(S),50
140 NEXT S
150 FOR S=8 TO 1 STEP -1
160 SOUND N(S),50
170 NEXT S
180 DATA 9394,8368,7456,7032
190 DATA 6269,5586,4976,4697
```

To minimize the processing time, the variables used by the program are defined as integers. In longer musical compositions, the time it takes the computer to execute the non-sound-generating statements can cause unwanted delays between tones. Using integer variables where appropriate can help reduce these delays.

The following example uses the **RND** function to produce an endless and ever-varying musical composition, which I call the "Cacophony Overture."

```
NEW
100 'Cacophony Overture
110 DEFINT P,L
120 A=RND(-VAL(RIGHT$(TIME$,2))-1)
130 P=INT(16383*RND(1)+1)
140 L=INT(25*RND(1)+1)
150 SOUND P,L
160 GOTO 130
```

SUMMARY

The **BEEP** statement produces a half-second tone each time it is executed. The **SOUND** statement produces a tone of varying pitch and duration. The general form of the **SOUND** statement is

SOUND *pitch,length*

The *pitch* parameter can range from 0 to 16838. The smaller *pitch* values produce treble tones and the larger *pitch* values produce bass tones. The *length* parameter can range from 0 to 250. Each increment of 50 in the *length* parameter represents approximately one second of duration.

The *pitch* values that correspond to musical tones are shown in Fig. 26-1.

The use of the Models 200 and 100 **SOUND ON** and **SOUND OFF** instructions (discussed in Lesson 25) does not affect the **BEEP** and **SOUND** *pitch,length* statements.

String Comparisons

Lesson 8 introduced the subject of relational tests and illustrated how they are used to compare numbers in decision-making instructions. Relational tests are not limited to numbers, however. Strings can be compared and tested just as easily.

A program that illustrates the use of string comparisons is shown below.

```
NEW
100 'String comparison example
110 INPUT "Fahrenheit or Celsius (F/C)";A$
120 IF A$="F" THEN GOSUB 150 'Calculate Fahrenheit
130 IF A$="C" THEN GOSUB 210 'Calculate Celsius
140 GOTO 110 'Loop
150 'Calculate Fahrenheit subroutine
160 INPUT "Enter degrees Celsius";CELSIUS
170 N=9/5*CELSIUS+32
```



```

180 GOSUB 270 'Round to 2 places
190 PRINT CELSIUS; "Celsius =";N;"Fahrenheit"
200 RETURN
210 'Calculate Celsius subroutine
220 INPUT "Enter degrees Fahrenheit";FAHRENHEIT
230 N=5/9*(FAHRENHEIT-32)
240 GOSUB 270 'Round to 2 places
250 PRINT FAHRENHEIT;"Fahrenheit =";N;"CELSIUS"
260 RETURN
270 'Rounding subroutine
280 N=INT(N*100+.5)/100
290 RETURN

```

Before running this program, look at lines 120 and 130. The purpose of these lines is to determine whether an "F" (for Fahrenheit) or a "C" (for Celsius) is entered in response to the "Fahrenheit or Celsius?" prompt. If an F is entered, the comparison at line 120 will be true and the subroutine at line 150 will be executed. If a C is entered, the comparison at line 130 will be true and the subroutine at line 210 will be executed. In either case, when execution returns from the subroutine, it proceeds to line 140, which sends the program back to line 110. If any character other than F or C is entered, both comparisons fail and again execution loops back to line 110.

A sample **RUN** is shown below.

```

RUN
Fahrenheit or Celsius (F/C)? C
Enter degrees Fahrenheit? 100
  100 Fahrenheit = 37.78 Celsius
Fahrenheit or Celsius (F/C)? F
Enter degrees Celsius? 232.78
  232.78 Celsius = 451 Fahrenheit

```

(Press **CTRL** **C** to halt program execution.)

If you ran this program and it did not accept your entry of an F or C, you were probably entering lowercase letters. Although it makes no difference whether you use lowercase or uppercase when enter-

ing BASIC commands or statements, the two cases are not considered equal by relational tests.

To understand how the computer distinguishes between uppercase and lowercase letters, you must know how string values are stored inside the computer. Internally, every string character is stored as a number between 0 and 255. These numbers are referred to as the *ASCII value* or *representation* of the character. ASCII is an acronym for American Standard Code for Information Interchange and defines a convention adopted by most computer manufacturers. A complete list of the ASCII values used to represent characters internally is given in Appendix A.

When the computer compares two string characters, it compares their ASCII values. If their ASCII values are identical, they are considered equal. Otherwise, they are considered unequal. In the case of f and F, the ASCII value of f is 102 and the ASCII value of F is 70. Therefore, they are not equal.

Because the computer uses ASCII values to make string comparisons, it can also test whether a string character is less than or greater than another string character. For example, a lowercase f is greater than an uppercase F ($102 > 70$). A table showing the meaning of the relational operators when used to compare string characters is given below.

Symbol	Meaning for String Comparisons
<	Precedes in ASCII order
>	Follows in ASCII order
=	Equal to
<>	Not equal to
<=	Equals or precedes in ASCII order
>=	Equals or follows in ASCII order

Comparisons can be performed on strings longer than one character. To do so, the computer compares the strings character by character, including any spaces in the string. (If you look at the list in

Appendix A, you will notice that the ASCII value of a "space" is 32. Therefore, a space is greater than any character with an ASCII value less than 32, and less than any character with an ASCII value greater than 32.)

The following program will help you understand string comparisons.

```
NEW
100 'String relation example
110 INPUT "String 1";S1$
120 INPUT "String 2";S2$
130 IF S1$<S2$ THEN PRINT S1$;"<";S2$
140 IF S1$=S2$ THEN PRINT S1$;"=";S2$
150 IF S1$>S2$ THEN PRINT S1$;">";S2$
160 GOTO 110
```

A sample **RUN** is shown below.

```
RUN
String 1? xyz
String 2? XYZ
xyz>XYZ
String 1? $
String 2? #
$>#
String 1? cat
String 2? car
cat>car
String 1? Bill
String 2? BILL
Bill>BILL
String 1? larger
String 2? large
larger>large
String 1? isotherm
String 2? isothermal
isotherm<isothermal
```

(Press **CTRL** **C** to halt program execution.)

This program displays the relationship between two strings. You should be able to explain the results of the first four comparisons by referring to the ASCII table in Appendix A. To understand the last two comparisons, you must know the following rule: *If two strings of unequal length are equal up to where the shorter string ends, the longer string is considered greater than the shorter string.* In effect, its extra length gives it a larger ASCII value. Therefore, "larger" is greater than "large" and "isothermal" is greater than "isotherm."

The null string, or string of zero length, is considered less than any other string:

```

RUN
String 1?  (Press ENTER without typing any
             characters.)
String 2?  .
<.
```

(Press **CTRL** **C** to halt program execution.)

The null string is entered by just pressing **ENTER**. Since it contains nothing, nothing is shown to the left of the < sign.

You may be wondering when the null string is useful. The null string provides a convenient method of checking if an entry is made in response to an **INPUT** prompt. Since it is easy to hit **ENTER** by mistake when responding to a prompt, it is a good idea to check for this error in any program that accepts the entry of string information. The next program provides an example of this type of error checking.

```

NEW
100 'Telephone memo pad program
110 RESTORE 190
120 INPUT "Name";N$
130 IF N$="" THEN 120
140 READ A$,P$
150 IF A$="XXX" THEN BEEP:PRINT "End of list":GOTO
    110
```

```

160 IF A$=N$ THEN PRINT A$;" ";P$:GOTO 110
170 GOTO 140 'Loop
180 'Start of DATA statements
190 DATA Steve,867-5309,Chris,521-0021
200 DATA Bob,522-2345,John,787-1964
210 DATA Nancy,212-322-7667,Ben,788-8828
220 DATA Ken,455-7223,Charles,787-9329 ext 231
999 DATA XXX,XXX

```

This program is a simplified electronic telephone directory. The names and numbers are stored in **DATA** statements, so new names and numbers can be added by simply entering **DATA** statements with line numbers less than 999. When the program is run, you are prompted to enter the name of the person whose phone number you want. The program searches through its *data list* looking for the entered name. If the name is found, the person's telephone number is displayed. Otherwise, the message "End of list" is displayed.

A sample **RUN** is shown below.

```

RUN
Name? Bob
Bob 522-2345
Name? charles
End of list
Name? Charles
Charles 787-9329 ext 231

```

(Press **CTRL** **C** to halt program execution.)

Notice that the program requires that you enter the name you are looking for exactly as it is listed in the **DATA** statements.

SUMMARY

The computer allows all relational tests to be performed on string values. The tests are based on a character-by-character comparison of the ASCII values of the strings. ASCII is an acronym for American

Standard Code for Information Interchange and defines a convention adopted by most computer manufacturers for representing string characters in a computer's memory. A complete list of the ASCII values used by the computer is given in Appendix A.

A table showing the meaning of the relational operators when used to compare string characters is given below.

Symbol	Meaning for String Comparisons
<	Precedes in ASCII order
>	Follows in ASCII order
=	Equal to
<>	Not equal to
<=	Equals or precedes in ASCII order
>=	Equals or follows in ASCII order

The ASCII values for lowercase letters are greater than the values for the corresponding uppercase letters.

The null string, or string of zero length, is considered to be less than any other string.

String Manipulations

In the telephone memo pad program developed in Lesson 27, you had to enter an entire name, correctly spelled and capitalized, if you wanted the program to find a telephone number. This requirement not only increased the difficulty of using the program, it placed on your memory the burden of exactly remembering the person's name. By using the **MID\$** function, you can place most of that burden on the computer's memory.

The **MID\$** function allows a program to divide a string into pieces, for either assignment or testing purposes. The general form of the **MID\$** function is

MID\$(string expression,position,length)

where *string expression* is the string constant or variable you wish to divide, *position* is the place in the string at which the segmenting is to begin, and *length* is the number of characters to split off.

The *length* parameter is optional. If the *length* parameter is omitted, the function returns the remainder of the string.

The following program will help you understand how this function works.

```

NEW
100 'MID$ demonstration program
110 TEST$="abcdefghijklmnopqrstuvwxyz"
120 INPUT "Enter starting position";P
130 INPUT "Enter length";L
140 PRINT MID$(TEST$,P,L)
150 GOTO 120

```

This program defines TEST\$ as the lowercase letters of the alphabet from a to z. It then displays a segment of that string based on the values you enter for starting *position* and *length*. A sample **RUN** is shown below.

```

RUN
Enter starting position? 1
Enter length? 12
abcdefghijkl
Enter starting position? 25
Enter length? 2
yz
Enter starting position? 11
Enter length? 7
klmnopq

```

(Press **CTRL** **C** to halt program execution.)

The first position in a string is 1, so if you enter a zero or negative number for starting position, a ?FC Error (illegal Function Call) will result when the **MID\$** function is executed. You can enter any nonnegative value that you want for length. If you enter a value that is greater than the number of characters from the starting position to the end of the string, the computer simply takes the rest of the string.

By using **MID\$**, you can modify the telephone memo pad program so that you only have to enter the first letter of a name and it will find all names beginning with that letter.

```

NEW
100 'Telephone memo pad program
110 RESTORE 190
120 INPUT "Name";N$
130 IF N$="" THEN 120
140 READ A$,P$
150 IF A$="XXX" THEN BEEP:PRINT "End of list":GOTO
    110
160 IF MID$(A$,1,1)=MID$(N$,1,1) THEN PRINT A$;" ";P$
170 GOTO 140
180 'Start of DATA statements
190 DATA Steve,867-5309,Chris,521-0021
200 DATA Bob,522-2345,John,787-1964
210 DATA Nancy,212-322-7667,Ben,788-8828
220 DATA Ken,455-7223,Charles,787-9329 ext 231
999 DATA XXX,XXX

```

The only change made to the original version was in line 160. This change causes the program to compare a segment of A\$ starting at position 1 with a length of 1 to a segment of N\$ starting at position 1 with a length of 1 (in other words, the first letter of A\$ with the first letter of N\$). If the two are equal, a match has been found and the name and phone number are displayed. Since there can be many names with the same first letter, the program continues to search the **DATA** statements until the end of the list, marked by the dummy values "XXX," is found.

A sample **RUN** is shown below.

```

RUN
Name: B
Bob   522-2345
Ben   788-8828
End of list

```

```

Name? C
Chris 521-0021
Charles 787-9329 ext 231
End of list

```

Press **CTRL** **C** to halt program execution.)

Since the program now looks only at the first letter of each string, it will display the names Chris and Charles even if you enter Cz for the name.

Another useful string function is the **LEN** function. The **LEN** function allows a program to determine how long a string is (that is, how many characters are in the string). The general form of the **LEN** function is

LEN(*string expression*)

where *string expression* can be a string constant, variable, or concatenated expression.

A program that demonstrates the **LEN** function is shown below.

```

NEW
100 'LEN demonstration program
110 INPUT "Enter sample string";S$
120 PRINT S$;" is";LEN(S$);"characters long"
130 S$="":GOTO 110

```

A sample **RUN** is shown below.

```

RUN
Enter sample string? test it
test it is 7 characters long
Enter sample string? (Press ENTER without
                      typing any characters.)
is 0 characters long

```

(Press **CTRL** **C** to halt program execution.)

The **LEN** function counts all characters in a string, including spaces. If a null string is entered, it has a length of zero.

The **STRING\$** function creates a string of a specific number of characters. The general form of the function is

STRING\$(length,character)

where *character* is a string constant, string variable, or ASCII value indicating the character to be repeated and *length* is the number of times it is to be repeated.

The following example demonstrates the operation of the **STRING\$** function.

```
NEW
100 'STRING$ demonstration program
110 INPUT "Enter sample character";CHAR$
120 INPUT "Enter string length";L
130 PRINT "The string is: ";STRING$(L,CHAR$)
140 GOTO 110
```

A sample **RUN** is shown below.

```
RUN
Enter sample character? I
Enter string length? 12
The string is: I I I I I I I I I I I I I I
Enter sample character? ##
Enter string length? 6
The string is: #####
```

(Press **CTRL** **C** to halt program execution.)

The **STRING\$** program displays L repetitions of the character that you enter. If you enter more than one character, only the first character is used to construct the string.

The maximum string length allowed by the computer is 255 characters. If you attempt to produce a string that is longer than this, the computer will display an ?FC Error (Function Call) message.

As indicated above, *character* can also be an ASCII value. To illustrate this feature, make the following changes to lines 110 and 130

```
110 INPUT "Enter ASCII value";A
130 PRINT "The string is: ";STRING$(L,A)
```

and **RUN** the program again.

```
RUN
Enter ASCII value? 122
Enter string length? 22
The string is: zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
Enter ASCII value? 64
Enter string length? 4
The string is: @@@@
```

(Press **CTRL** **C** to halt program execution.)

Refer to Appendix A for a list of the ASCII values. As indicated in the appendix, ASCII values less than 32 are assigned to nondisplayable characters.

Observe that this program does not actually assign the result of the **STRING\$** function to a string variable. When desirable, this can be done by an assignment statement such as

```
TEST$=STRING$(5,MID$(C$,3,1))
```

Notice that a string function can be used as the argument of another string function. This type of construction is valid as long as the inner function produces the kind of result expected by the outer function. In this example, the **MID\$** function produces a string value for the *character* parameter of the **STRING\$** function.

The next program puts these functions to work. This program provides a simple version of a word guessing game. The game requires two players. The first player enters a word while the second player is not looking. The second player must discover the word by guessing which letters are in the word. Every time a correct guess is made, that letter is put in the appropriate place(s) in the blank word.

The computer keeps a count of how many guesses were made and displays that number at the end of each round.

```

NEW
100 'Guess word program
110 INPUT "Enter secret word";W$
120 IF W$="" THEN 110
130 CLS 'Clear screen
140 L=LEN(W$)
150 GUESS=0 'Number of guesses made
160 CL=0 'Number of correct letters
170 FMT$=STRING$(L,"-")
180 'Begin guessing loop
190 IF CL=L THEN 360 'Round finished
200 PRINT FMT$;" ";GUESS
210 INPUT "Your Guess";GUESS$
220 IF GUESS$="" THEN 210
230 GUESS=GUESS+1
240 'Begin string search loop
250 FOR P=1 TO L
260 IF MID$(W$,P,1)=GUESS$ THEN GOSUB 300
270 NEXT P
280 GOTO 190
290 'Put letter in FMT$ subroutine
300 L$=MID$(FMT$,1,P-1)
310 R$=MID$(FMT$,P+1,L-P+1)
320 FMT$=L$+GUESS$+R$
330 CL=CL+1
340 RETURN
350 'Round finished
360 BEEP
370 PRINT "You guessed ";W$;" in";GUESS;"guesses"
380 GOTO 110

```

A sample **RUN** is shown below.

```

RUN
Enter secret word? help

```

```

----- 0
Your guess? h
h--- 1
Your guess? z
h--- 2
Your guess? p
h--p 3
Your guess? e
he-p 4
Your guess? l
You guessed help in 5 guesses

```

(Press **CTRL** **C** to halt program execution.)

A brief explanation of the program follows.

The program begins by prompting for the secret word and storing that word in W\$. Then the display is cleared to prevent the second player from seeing the secret word. The **LEN** function is used to determine how long the secret word is. After setting the variables GUESS and CL to zero, the program uses the value produced by the **LEN** function to create a format string (FMT\$) showing how long the secret word is. The **STRING\$** function is used for this purpose since it allows a variable to set the string length.

Line 180 begins the guessing loop of the program. The loop first checks if the number of correct letters (CL) equals the number of letters in the secret word (L). If this test is true, the secret word has been guessed and the round is over.

If CL does not equal L, the program proceeds to prompt for a guess. The variable GUESS is increased each time a guess is made so that the program can display the number of guesses at the end of the round.

The string search part of the program is a loop that uses the **MID\$** function to check each of the letters in the secret word to determine if a correct guess has been made. When a correct guess is found, the subroutine at line 290 puts the letter in the correct location in

the format string, increases the value of CL by one, and returns execution to the string search loop. After every letter in the string has been checked, execution is transferred to line 190 to begin the guessing loop again.

SUMMARY

The **MID\$** function is used to split a string into a segment or piece. The general form of the function is

MID\$(*string expression*,*position*,*length*)

where *string expression* is the string to be split, *position* is the place where the segmenting is to begin, and *length* is the number of characters to split off. The first character in the string is position 1.

The **LEN** function determines the number of characters in a string. The general form of the function is

LEN(*string expression*)

where *string expression* is the string whose length is to be tested.

The **STRING\$** function is used to create a string that consists of a certain number of repetitions of a character. The general form of the function is

STRING\$(*length*,*character*)

where *character* is a string constant, string variable, or ASCII value and *length* is the number of times *character* is to be repeated.

The ASCII Code Functions

As discussed in Lesson 27, string comparisons are based on a character-by-character matching of the ASCII values of the strings. If the ASCII values are identical, the strings are considered equal. Otherwise, they are considered unequal.

String comparisons provide an indirect glimpse of ASCII values. These values can be viewed directly using the **ASC** function. The **ASC** function has the general form

ASC(string expression)

The **ASC** function returns the ASCII value of the first character of the *string expression*.

A program that demonstrates the **ASC** function is shown below.

NEW

100 'ASC demonstration program


```

110 INPUT "Sample string";S$
120 FOR C=1 TO LEN(S$)
130 PRINT ASC(MID$(S$,C,1));
140 NEXT C
150 PRINT
160 GOTO 110

```

RUN the program and observe the results.

```

RUN
Sample string? test it
 116 101 115 116 32 105 116
Sample string? TEST IT
 84 69 83 84 32 87 84
Sample string? 1234
 49 50 51 52

```

(Press **CTRL** **C** to halt program execution.)

The program dissects the strings that you enter into their ASCII values. This is accomplished by using a **FOR TO NEXT** loop and the **MID\$** function to split the string into individual characters. The **ASC** function is then used to determine the ASCII value of the characters. (The **PRINT** statement in line 150 is used to cancel the pending print condition, created by the semicolon in line 130.)

Note that the **ASC** function demands a string argument. If you use a numeric argument (not a string of digits as in the example), the computer displays a ?TM Error (Type Mismatch) message.

If you know the ASCII value of a character, you can generate the string of that character with the

CHR\$ (*numeric expression*)

function. The **CHR\$** function is the opposite of the **ASC** function. It takes a numeric value between 0 and 255 and returns the string character represented by that value.

A quick demonstration of the function is shown below.

```

PRINT CHR$(97)
a
Ok
PRINT CHR$(126)
~
Ok
PRINT CHR$(33);CHR$(43);CHR$(53)
!+5
Ok

```

If you look at the table in Appendix A, you will see that ASCII values 0 through 31 are not assigned to displayable characters. Instead, many of these values are assigned special functions. For example, **PRINTing CHR\$(7)** produces the **BEEP** tone.

One of the most useful ASCII values below 32 is 13. The ASCII value 13 produces the same result as pressing the **ENTER** (or **↵**) key. Thus, concatenating **CHR\$(13)** to a function key definition adds **ENTER** to the key definition. For example,

```
KEY 6,"BEEP"+CHR$(13)
```

assigns the sequence **BEEP ENTER** to function key 6. The **CHR\$(13)** in the key definition eliminates the need to press **ENTER** after pressing **F-6**.

Here's another interesting application of the **CHR\$** function.

```

NEW
100 INPUT "Sample string";S$
110 PRINT CHR$(27);CHR$(112);S$
120 PRINT CHR$(27);CHR$(113);
130 GOTO 100

```

When you **RUN** this program, the computer displays the sample string that you enter in "reverse video." The ASCII values 27 and 112 turn reverse video on. ASCII values 27 and 113 turn it off.

One of the main advantages of the **ASC** and **CHR\$** functions is that they enable a program to manipulate string values numerically.

The following program demonstrates how this can be done.

```
NEW
100 'Encrypt/decrypt example
110 INPUT "Encrypt/Decrypt (E/D)";A$
120 IF A$="E" THEN GOSUB 150
130 IF A$="D" THEN GOSUB 230
140 GOTO 110
150 'Encrypt subroutine
160 INPUT "String to encrypt";E$
170 E1$=""
180 FOR C=1 TO LEN(E$)
190 E1$=E1$+CHR$(ASC(MID$(E$,C,1))+5)
200 NEXT C
210 PRINT E1$
220 RETURN
230 'Decrypt subroutine
240 D1$=""
250 INPUT "String to decrypt";D$
260 FOR C=1 TO LEN(D$)
270 D1$=D1$+CHR$(ASC(MID$(D$,C,1))-5)
280 NEXT C
290 PRINT D1$
300 RETURN
```

The critical parts of this program are the **FOR NEXT** loops in lines 180–200 and 260–280. The first **FOR NEXT** loop encrypts the string that you enter. This is accomplished by splitting the string into individual characters (using **MID\$**), calculating the ASCII value of each character (using **ASC**), adding 5 to that value, and then converting the resulting ASCII value back to a string character (using **CHR\$**). The characters are recombined into a single string (E1\$) by concatenating them together.

The second **FOR NEXT** loop decrypts the string by performing the identical process, but subtracting 5 from the ASCII value of each character instead of adding 5.

A sample **RUN** of the program is shown below.

```

RUN
Encrypt/Decrypt (E/D)? E
String to encrypt? Mississippi
Rnxxnxxnuun
Encrypt/Decrypt (E/D)? D
String to decrypt? Rnxxnxxnuun
Mississippi
Encrypt/Decrypt (E/D)? D
String to decrypt? Yt%9j%tw%st9%9t%9jD
To be or not to be?

```

(Press **CTRL** **C** to halt program execution.)

SUMMARY

BASIC provides two functions for dealing directly with the ASCII value of string characters. The **ASC** function has the general form

ASC (*string expression*)

and returns the ASCII value of the first character of *string expression*. The value returned is a number between 0 and 255 that can be manipulated mathematically by a program.

The opposite of the **ASC** function is the **CHR\$** function. The **CHR\$** function has the general form

CHR\$ (*numeric expression*)

and returns the string character represented by *numeric expression*. The value of *numeric expression* must be between 0 and 255.

As indicated by Appendix A, many ASCII values less than 32 are assigned to special functions. A list of the special ASCII values illustrated in this lesson is given below.

ASCII Values	Action
7	Produces BEEP tone
13	Produces a "carriage return" (see note below)
27 112	Turn "reverse video" on
27 113	Turn "reverse video" off

A "carriage return" produces the same effect as pressing the **ENTER** (or ) key.

Numeric/String Conversions

A string value, even one consisting of all digits, cannot be used as a numeric value. This restriction is not absolute, however, because BASIC provides a special function for converting a string of digits to a numeric value. The instruction for converting string data to numeric data is

VAL (*string expression*)

The **VAL** function requires a string argument. Using a numeric argument will provide a ?TM Error (Type Mismatch) message.

A program that demonstrates the **VAL** function is shown below.

```
NEW
100 'VAL demonstration program
110 INPUT "Enter a string value";S$
120 PRINT VAL(S$)
130 GOTO 110
```

A sample **RUN** is shown below.

```

RUN
Enter a string value? 1.0046296296
1.0046296296
Enter a string value? #9
0
Enter a string value? 1807 Primrose
1807
Enter a string value? 3313 30th Street
331330
Enter a string value? -1 . 2 3
-1.23
Enter a string value? testing. . . 1 2 3
0
    
```

(Press **CTRL** **C** to halt program execution.)

Observe that the **VAL** function ignores leading spaces and spaces between digits. But as soon as a character other than a digit, space, plus sign, minus sign, or decimal point is encountered, the conversion process stops. The numeric value accumulated to that point is returned.

Technical Note: The **VAL** function will accept an E or D as a convertible character in string constructions that follow the rules of exponential (scientific) notation. For example, the string 4.552E-40 will convert to 4.552E-40 (a single-precision number in exponential format).

The **STR\$** function performs the inverse of the **VAL** function. It converts a numeric value to string format. The general form of the **STR\$** function is

STR\$(numeric expression)

The **STR\$** function returns a string of digits equal to the numeric value of *numeric expression*.

A demonstration of the **STR\$** function is shown below.

```
PRINT STR$(18700)
18700
Ok
PRINT "THIS IS A STRING:"+STR$
(-1.00000008)
THIS IS A STRING:-1.00000008
Ok
```

Note that the string produced by the **STR\$** function contains a leading space for positive values.

SUMMARY

BASIC provides two special functions for converting string data to numeric data, and vice versa. The

VAL(*string expression*)

function converts a string of digits to a numeric value. The function ignores leading spaces and spaces between digits. The conversion process stops when a character other than a digit, decimal point, plus sign, or minus sign is encountered. The numeric value accumulated to that point is returned.

The

STR\$(*numeric expression*)

function converts a numeric value to a string value. The **STR\$** function is the opposite of the **VAL** function.

REVIEW TEST 6

1. Which of the following are valid BASIC statements?

- (a) 100 IF D="6" THEN BEEP:PRINT "DONE!"
- (b) 340 T\$=STRING\$("*",15)
- (c) 110 IF LEN(P\$)=STRING\$(7,S\$) THEN 100
- (d) 400 R=LEN(MID\$(T\$,X,Y))
- (e) 501 NW\$=STRING\$(3,76)+MID\$(LD\$,3,2)
- (f) 110 SOUND 7900,300
- (g) 220 AA\$=MID\$("1234567",3,99)
- (h) 300 PRINT ASC(VAL("S"))
- (i) 310 PRINT VAL(ASC("S"))
- (j) 660 LL\$=STRING\$(257,84)
- (k) 500 U=VAL(CHR\$(49)+CHR\$(50)+CHR\$(51))
- (l) 100 PRINT LEN(MID\$(AA\$,2)+STRING\$(88,ASC("*")))
- (m) 950 IF ASC(B\$)=7 THEN PRINT CHR\$(7)
- (n) 700 E\$=CHR\$(27)+CHR\$(112)+"** ERROR **"
- (o) 200 PRINT ASC(STR\$(VAL(MID\$(CHR\$(48)+"123",2))))

2. What is an ASCII value, and why is it important in programming?

3. Which of the following string comparisons are true?

- (a) "Bill Smith" < "Bill Smithe"
- (b) "6" < "5"
- (c) "ASCII" > "ascii"
- (d) "null" <> " "
- (e) "<" < ">"
- (f) "Store1" >= "Store2"
- (g) "dysphemia" > "dysphonia"
- (h) "*****" = "*****"
- (i) "#!\$%&" > "#'\$%&"
- (j) "phantasmagoria" = "phantas" + "magoria"

4. What is the maximum length of a string?

5. The English alphabet is made up of 21 consonants and five vowels. Write a program that reads the consonants into an array named C\$ and the vowels into an array named V\$. Then have the program construct and display random words consisting of a consonant, a vowel, and a consonant.

6. Write a program that allows you to input a string value and then displays the string in "reverse" order (last character first, next-to-last character second, etc.). *Hint:* Use the **MID\$** function to take the string apart one character at a time, beginning with the last character in the string, and use concatenation to rebuild the string.

7. Write a program that converts any lowercase letters that you input to uppercase letters. *Hint:* Perform any adjustments on the ASCII value of each character.

8. Write a program that requests the capital city of the countries listed below, and then checks the answer that is entered to see if it is correct.

Country	Capital
Afghanistan	Kabul
Brazil	Brasilia
Egypt	Cairo
England	London
Ethiopia	Addis Ababa
France	Paris
Greece	Athens
India	New Delhi
Japan	Tokyo
Mexico	Mexico City
United States	Washington
U.S.S.R.	Moscow

Reserving String Space

As noted in Lesson 4, your computer automatically reserves 256 bytes of memory for storing string data when you select BASIC from the Main Menu screen. If you exceed this limit, the computer will display an ?05 Error (Out of String space) message. This limitation is easily demonstrated:

```
NEW
S$=STRING$(255,"*")
T$="12"
?05 Error
```

256 bytes provide enough memory space to store 256 string characters. In the sequence shown above, S\$ is set to 255 asterisks (the maximum string length). When an attempt is made to assign two additional string characters to T\$, the 256-character limit is exceeded and the computer displays the out-of-string-space error message.

Obviously, BASIC would be woefully inadequate if it provided no method of increasing the amount of memory devoted to string data. (Although 256 bytes of string memory is sufficient for the examples in this book, you can easily exceed this limit when you begin to create programs of your own.) The BASIC instruction for reserving memory for string data is the **CLEAR** statement. The **CLEAR** statement has the general form

CLEAR *string space*

The *string space* parameter specifies the number of bytes of memory to reserve for string storage.

Technical Note: The **CLEAR** statement permits the use of an optional *high memory* parameter. The purpose of this parameter is to reserve memory for machine language subroutines, a subject that is not discussed in this book.

Whenever you use a **CLEAR** statement in a program, you should locate the statement at the beginning of the program. The reason for this requirement is that the **CLEAR** statement has the “side effect” of *erasing the value of all variables*, both string and numeric. The **CLEAR** statement also cancels any **DIM** statements that have been executed.

By carefully selecting the amount of memory to reserve for string space, you can increase the memory efficiency of your programs.

Determining the correct amount of string memory is often a trial-and-error process. A “ballpark” figure is selected initially. If during the testing of the program, an **?05 Error** occurs, this value is increased. On the other hand, if the error does not occur, the amount of memory **CLEARed** is reduced slightly. Eventually, a value just larger than the minimum is arrived at.

Most of the guesswork can be eliminated from this process by a judicious use of the **FRE** function. The **FRE** function has two forms:

FRE(*string expression*)
FRE(*numeric expression*)

The **FRE**(*string expression*) form of the function returns the amount of string memory that is unused. The **FRE**(*numeric expression*) form of the function returns the total amount of unused memory less the amount of memory reserved for string space. Thus, the argument type determines the kind of information returned.

To illustrate these functions, exit the BASIC workspace and return to the Main Menu screen. Then reenter BASIC. Note the number of bytes of free memory reported by the computer. This value represents the amount of random access memory in your computer less the memory used for RAM files and the 256 bytes of memory reserved for string space. The value returned by the **FRE**(*numeric expression*) function should match this value:

```
PRINT FRE(0)
10867
Ok
```

The value returned by your computer will differ from the value shown above, but it should match the value displayed when you selected BASIC. In the example, the function argument is 0. You can use any numeric constant or variable; the result returned will be the same.

```
PRINT FRE(N)
10867
Ok
```

Now display the amount of unused string space:

```
PRINT FRE(" ")
256
Ok
```

No string variable assignments have been made, so all the reserved string space is available. The string argument used was the null string. Of course, you could have used any string constant or variable.

Increase the amount of string space to 456 bytes.

```
CLEAR 456
Ok
PRINT FRE(A$)
456
Ok
```

All the reserved string space remains unused. Reenter the assignment statements given at the beginning of the lesson and measure the available string space again.

```
S$=STRING(255,"*")
Ok
T$="12"
Ok
PRINT FRE(" ")
199
Ok
PRINT 456-255-2
199
Ok
```

As expected, the amount of available string space has been reduced by 257 bytes. As a final experiment, enter the following instructions.

```
NEW
PRINT FRE(" ")
456
Ok
```

The **NEW** command does not alter the amount of string space, even though it does erase the value of all variables. The memory reserved for storing string data will remain unchanged until you alter it with another **CLEAR** statement or until you leave the BASIC workspace.

SUMMARY

The computer automatically reserves 256 bytes of memory for storing string data when BASIC is selected from the Main Menu screen.

This is enough memory to store 256 string characters. If more string space is needed, it can be allocated using the **CLEAR** statement. The **CLEAR** statement has the general form

CLEAR *string space*

The *string space* parameter specifies the amount of memory to reserve.

Executing the **CLEAR** statement erases the value of all variables and cancels the effect of any **DIM** statements. For this reason, the **CLEAR** statement, when needed, is generally placed at the beginning of a program. Executing a **NEW** statement does not alter the amount of memory reserved for string space.

The amount of string memory that is unused at any given moment can be checked using the **FRE** function with a string argument. Using a numeric argument with the **FRE** functions returns the total amount of unused RAM less the amount of memory reserved for string space.

The two forms of the **FRE** function are

FRE(*string expression*)
FRE(*numeric expression*)

The argument used with the **FRE** function can be a constant or a variable.

LINE INPUT, INPUT\$, and INKEY\$

The **INPUT** statement introduced in Lesson 6 provides a method of entering keyboard information into a program. The **INPUT** statement can display a prompt indicating what information to enter and can assign input to either string or numeric variables. If desirable, the statement can even assign input to multiple variables.

For some applications, however, the **INPUT** statement is not entirely satisfactory. For example, if you are writing a mailing list program and want to input names in last-name, first-name order, with a comma dividing the last and first names, the **INPUT** statement cannot be used. (A comma is used to separate information to be assigned to multiple variables by an **INPUT** statement.) Another aspect of the **INPUT** statement that can be undesirable in some circumstances is the requirement that the **ENTER** key be pressed to end the entry of data.

The three instructions discussed in this lesson provide alternative methods of accepting keyboard information. Each instruction has

advantages and disadvantages, and applications for which it is best suited. Of the three instructions, the one that most closely duplicates the action of the **INPUT** statement is the **LINE INPUT** statement.

The **LINE INPUT** statement has the general form

LINE INPUT *prompt*;string variable

The *prompt* parameter is optional and follows the same rules as the *prompt* parameter of the **INPUT** statement. Unlike the **INPUT** statement, however, **LINE INPUT** accepts only string data. If the *prompt* parameter is used, it must be separated from the input variable by a semicolon.

Besides accepting only string input, the **LINE INPUT** statement differs in the following ways:

- The statement permits you to enter commas, quotes, and leading spaces.
- The statement does not display a question mark prompt, even when executed without a *prompt* parameter.
- The statement does not retain the previous value of the input variable when only the **ENTER** key is pressed. (When you press **ENTER** without typing any characters in response to a statement such as **INPUT "Enter value";S\$,** the computer retains the previous value of S\$, if any. When you press **ENTER** in response to the statement **LINE INPUT "Enter value";S\$,** the computer assigns the null string to S\$.)

As with the **INPUT** statement, the **LINE INPUT** statement requires that you press **ENTER** to signal the end of keyboard input.

A program that demonstrates the **LINE INPUT** statement is shown below.

```
NEW
100 'LINE INPUT example
110 LINE INPUT "Enter a string: ";S$
120 PRINT S$
130 GOTO 110
```

A sample **RUN** is shown below.

```

RUN
Enter a string: Nimzowitsch, Aron
Nimzowitsch, Aron
Enter a string: "Every stick has two
ends."
"Every stick has two ends."
Enter a string:          1,000,000,000
                  1,000,000,000

```

(Press **CTRL** **C** to halt program execution.)

Obviously, the **INPUT** and **LINE INPUT** statements do not differ that greatly. You may be more surprised at the operation of the **INPUT\$** function.

The **INPUT\$** function instructs the computer to accept a specific number of characters from the keyboard. The function has the general form

INPUT\$(numeric expression)

where *numeric expression* defines the number of keyboard characters to accept, up to 255 characters.

Enter the following demonstration program.

```

NEW
100 'INPUT$ example
110 PRINT "Enter a five-character password:";
120 A$=INPUT$(5)
130 IF A$<>"BASIC" THEN BEEP:PRINT:GOTO 110

```

When you **RUN** this program, you are prompted for a five-character password. Try a number of different five-character combinations before you enter the word expected by the program. Note that the **INPUT\$** function counts the **ENTER**, **BKSP**, and cursor movement keys as characters.

Clearly, the **INPUT\$** function differs dramatically from the **INPUT** and **LINE INPUT** statements. Although the function does display

a flashing cursor, it does not display the characters read from the keyboard. Nor does it use **ENTER** to signal the end of input; instead, it waits until you have typed the number of characters specified by the argument before it allows execution to continue.

The final instruction discussed in this lesson, the **INKEY\$** function, can be thought of as an **INPUT\$(1)** instruction that does not wait for a key to be pressed. The **INKEY\$** function performs an immediate scan of the keyboard. If a key is pressed while the function is executing, or if a keystroke is waiting in the "type ahead" buffer, the value of the key is returned by the function. Otherwise, a null string is returned by the function. In addition to not waiting for a key to be pressed, the **INKEY\$** function does not display a flashing cursor.

The following program demonstrates the **INKEY\$** function.

```
NEW
100 'INKEY$ demonstration
110 A$=INKEY$
120 IF A$="" THEN PRINT "-"; ELSE BEEP:PRINT "+";
130 GOTO 110
```

Note that the **INKEY\$** function requires no argument.

When you **RUN** this program, the computer madly displays hyphens. Each hyphen indicates that the **INKEY\$** function has scanned the keyboard once and found no key depressed. The **INKEY\$** function is very quick, as demonstrated by the speed at which the hyphens are displayed. When you press a key the computer beeps and displays a plus sign.

Stop the program by pressing **CTRL C**. Then make the following changes to line 120.

```
120 IF A$="" THEN GOTO 110 ELSE PRINT ASC(A$);
```

This modification instructs the computer to display the ASCII value of any key that you press. With this program, you can confirm that the **ENTER**, cursor movement, and **CTRL** key combinations return string characters with ASCII values matching the values given in Appendix A. The following table lists a few of the keys that you

may wish to check. The **INPUT\$** and **INKEY\$** functions provide a program with a method of checking for these special keystrokes.

Key Sequence	ASCII Value
CTRL A	1
BKSP (BS)	8
TAB	9
ENTER	13
CTRL Z	26
ESC	27
→	28
←	29
↑	30
↓	31

For examples of practical uses of the **INPUT\$** function, refer to Lessons 34, 35, and 36.

SUMMARY

The **LINE INPUT**, **INPUT\$**, and **INKEY\$** instructions provide alternatives to the **INPUT** statement for entering keyboard information into a program. Of the three instructions, the **LINE INPUT** statement most closely duplicates the action of the **INPUT** statement.

The **LINE INPUT** statement has the general form

LINE INPUT *prompt;string variable*

The *prompt* parameter can be a string constant or a string variable, and is optional. The input variable must be a string variable.

The **LINE INPUT** statement differs from the **INPUT** statement in the following ways.

- The statement permits the entry of commas, quotes, and leading spaces.
- The statement does not display a question mark prompt.

- The statement does not retain the previous value of the input variable when only the **ENTER** key is pressed.

The **INPUT\$** function has the general form

INPUT\$(numeric expression)

where *numeric expression* defines the number of characters to accept from the keyboard, up to a limit of 255 characters.

The **INPUT\$** function does not display the characters read by the function, nor does it require that **ENTER** be pressed to end input. The computer waits until the number of characters specified by the argument have been input before execution continues.

The **INKEY\$** function performs an immediate scan of the keyboard. If a key is pressed while the function is executing, or if a keystroke is waiting in the "type ahead" buffer, the string value of the key is returned by the function. If no key is pressed or waiting in the buffer, the function returns a null string. Thus, the **INKEY\$** function returns a key value if one is available when the function is executed, but does not wait for one if no key is depressed.

The **INKEY\$** function does not have an argument.

The **INPUT\$** function displays a flashing cursor when executed; the **INKEY\$** function does not.

Logical Operations

In a decision-making statement such as

```
IF DAY>31 THEN GOSUB 200
```

the computer decides to execute or not execute the subroutine based on the result of the relational test `DAY>31`. If the relation is true, the subroutine call is executed. If the relation is false, the subroutine call is skipped.

Decisions can also be based on the results of logical operators. Logical operators allow several true-false conditions to be tested with a single **IF THEN** statement. The logical operators are **AND**, **OR**, **XOR**, **EQV**, **IMP**, and **NOT**.

The **AND** operator compares two separate true-false conditions and arrives at a single true or false result, based on the rules shown in the table below. The results of the individual true-false tests (repre-

sented by X and Y) are shown on the left and the combined result is shown on the right.

X	Y	X AND Y
True	True	True
True	False	False
False	True	False
False	False	False

As you can see from studying the table, for the result of an **AND** operation to be true, both individual tests must be true. If either of the individual tests is false, the combined result is false.

An example of an **AND** decision-making test is shown below.

```
100 IF MO=2 AND DAY=31 THEN PRINT "Invalid day"
```

If MO equals 2 and DAY equals 31, the message will be displayed. If either or both variables have some other value, the **PRINT** statement will be skipped.

The rules controlling the operation of the **OR** operator are shown below.

X	Y	X OR Y
True	True	True
True	False	True
False	True	True
False	False	False

The result of an **OR** operation is false only when both individual results are false. If either or both individual results are true, the combined result is true also. An example of an **OR** test is shown below.

```
100 IF DAY<1 OR DAY>31 THEN PRINT "Invalid day"
```

If day is less than 1 or greater than 31, the message will be displayed.

The rules controlling the operation of the **XOR** (exclusive **OR**) operator are shown below.

<u>X</u>	<u>Y</u>	<u>X XOR Y</u>
True	True	False
True	False	True
False	True	True
False	False	False

The result of an **XOR** test is true only when the results of the individual tests differ. Whenever the individual tests have identical results, the combined result is false.

An example of an **XOR** test is shown below.

```
100 IF A=5 XOR B=10 THEN GOSUB 200
```

If either A or B equals the tested value, but not both, the subroutine call is executed. If both are true, or both are false, the subroutine call is skipped.

The rules controlling the operation of the **EQV** (equivalence) operator are shown below.

<u>X</u>	<u>Y</u>	<u>X EQV Y</u>
True	True	True
True	False	False
False	True	False
False	False	True

The result of an **EQV** test is true when the individual tests have identical results. When the individual tests have different results, the combined result is false. (This is the opposite of the **XOR** operator.)

An example of an **EQV** test is shown below.

```
100 IF A=5 EQV B=10 THEN BEEP
```

If both tests are true or both tests are false, the computer **BEEPs**. If one test is true and one test is false, the **BEEP** statement is skipped.

The rules controlling the operation of the **IMP** (implication) operator are shown below.

<u>X</u>	<u>Y</u>	<u>X IMP Y</u>
True	True	True
True	False	False
False	True	True
False	False	True

The result of an **IMP** test is false when the first test is true and the second test is false. For all other cases, the combined result is true.

An example of the **IMP** operator is shown below.

```
100 IF I=7 IMP J<>11 THEN END
```

If $I=7$ and $J=11$, the **END** statement is skipped. Otherwise, the **END** statement is executed.

The last logical operator is the **NOT** operator. The **NOT** operator reverses the result of a single true-false test.

<u>X</u>	<u>NOT X</u>
True	False
False	True

An example of the **NOT** operator is shown below.

```
100 IF NOT A=5 THEN GOSUB 200
```

If A equals 5, the subroutine call is skipped. If A does not equal 5, the subroutine call is executed.

The logical operators can be used to create very sophisticated decision-making tests: for example,

```
IF A=5 AND B<10 OR C>15 AND NOT D=20 THEN X=2
```

When different logical operators are combined in one decision-making test, the operators are evaluated from left to right in the following order:

Levels of Precedence for Logical Operators

NOT (highest priority)
AND
OR
XOR
EQV
IMP (lowest priority)

The order in which the decision-making example given above will be evaluated is illustrated in Fig. 33-1. The first operator performed is the **NOT** operator; the last operator performed is the **OR** operator.

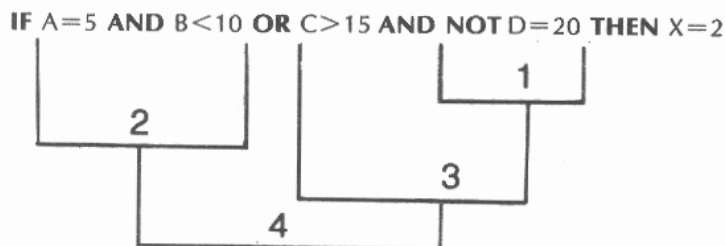


Fig. 33-1 Example illustrating the order of evaluation of multiple logical operators.

If you want logical operators to be evaluated in another order, you can place parentheses around portions of the test, just as in mathematical calculations. For example, entering

IF A=5 AND (B<10 OR C>15) THEN X=2

will force the computer to evaluate the **OR** operator before the **AND** operator.

The logical operators allow programs to be shortened by combining several **IF THEN** tests into one statement. The following program demonstrates how convenient this can be. The purpose of this program is to ensure that values entered for month, day, and year are valid. Without the capability to use logical operators in these tests, the program would require many more **IF THEN** statements.

```

NEW
100 'Validate date program
110 INPUT "Enter month";MO
120 IF MO<1 OR MO>12 OR MO<>INT(MO) THEN 210
130 INPUT "Enter day";DAY
140 IF DAY<1 OR DAY>31 OR DAY<>INT(DAY) THEN
    230
150 IF DAY=31 AND (MO=4 OR MO=6 OR MO=9 OR
    MO=11) THEN 230
160 IF DAY>29 AND MO=2 THEN 230
170 INPUT "Enter year"; YEAR
180 IF YEAR/4<>INT(YEAR/4) AND MO=2 AND DAY=29
    THEN 250
190 PRINT MO;" / ";DAY;" / ";YEAR;" is a valid date"
200 GOTO 110 'Loop
210 PRINT "Invalid month"
220 GOTO 110
230 PRINT "Invalid day"
240 GOTO 130
250 PRINT YEAR;" is not a leap year"
260 PRINT "so";DAY;" is an invalid day"
270 GOTO 130

```

The functions of the decision-making tests are explained below.

- Line 120 Rejects the entered month if it is less than 1 or greater than 12 or a fractional value
- Line 140 Rejects the entered day if it is less than 1 or greater than 31 or a fractional value
- Line 150 Rejects the entered day if it is equal to 31 and any one of the following months was entered: April, June, September, or November
- Line 160 Rejects the entered day if it is greater than 29 and the month is February
- Line 180 Rejects the entered day if the year is not a leap year and the month is February and the day is equal

to 29 (see Lesson 13 for a discussion of the leap-year test)

A sample program **RUN** appears below.

```

RUN
Enter month? 2
Enter day? 29
Enter year? 1981
  1981 is not a leap year
so 29 is an invalid day
Enter day? 29
Enter year? 1980
2 / 29 / 1980 is a valid date
Enter month? 7
Enter day? 31
Enter year? 1949
7 / 31 / 1949 is a valid date

```

(Press **CTRL** **C** to halt program execution.)

SUMMARY

Logical operators allow several true-false conditions to be tested with a single **IF THEN** statement. The logical operators are **AND**, **OR**, **XOR**, **EQV**, **IMP**, and **NOT**.

The **AND**, **OR**, **XOR**, **EQV**, and **IMP** operators produce a single true or false result from a comparison of two separate decision-making tests. A table giving the rules governing these operations is shown below (X and Y represent the results of the individual tests).

X	Y	X AND Y	X OR Y	X XOR Y	X EQV Y	X IMP Y
True	True	True	True	False	True	True
True	False	False	True	True	False	False
False	True	False	True	True	False	True
False	False	False	False	False	True	True

In summary, **AND** is true when both individual tests are true, **OR** is true when at least one individual test is true, **XOR** is true when the results of the individual tests differ, **EQV** is true when the individual tests have identical results, and **IMP** is true in all cases except when the first test is true and the second test is false.

The **NOT** operator reverses the status of a single decision-making test, as shown in the table below.

<u>X</u>	<u>NOT X</u>
True	False
False	True

The ON GOTO and ON GOSUB Statements

The **ON GOTO** and **ON GOSUB** statements are decision-making statements that base decisions on a numeric value and not on the result of a true-false condition like the **IF THEN** statement. These statements have the general form

ON *numeric expression* GOTO *line list*
ON *numeric expression* GOSUB *line list*

where *numeric expression* is a numeric variable or calculation and *line list* is a list of line numbers separated by commas.

The value of *numeric expression* determines which of the transfer locations in the *line list* is selected. If *numeric expression* equals 1, the program transfers to the first line number; if 2, to the second line number; if 3, to the third line number, and so on. If *numeric expression* has a value that is less than 0, a ?FC Error (illegal Function Call) condition occurs. If *numeric expression* has a value

that is greater than the number of line numbers in the *line list* or a value of 0, the **ON GOTO** or **ON GOSUB** statement is skipped and execution proceeds with the next statement in the program.

A sample **ON GOTO** statement is shown below.

```
50 ON N GOTO 100,200,300
```

When this statement is executed, the computer will transfer program execution to line 100 if $N=1$, to line 200 if $N=2$, and to line 300 if $N=3$. A single **ON GOTO** statement such as this can replace a three-line program sequence such as

```
50 IF N=1 THEN 100
60 IF N=2 THEN 200
70 IF N=3 THEN 300
```

The **ON GOSUB** statement functions identically to the **ON GOTO** statement except that the line numbers in the *line list* are treated as subroutines rather than direct transfers. For example, when

```
250 ON K/2+1 GOSUB 3000,205,3100,2000
```

is executed, the program calls line 3000 if $K/2+1=1$, line 205 if $K/2+1=2$, line 3100 if $K/2+1=3$, and line 2000 if $K/2+1=4$. When the program returns from the selected subroutine, execution continues with the first line following the **ON GOSUB** statement.

A program that uses the **ON GOTO** statement is shown below.

```
NEW
100 'Meters/Liters/Grams conversion program
110 PRINT "1-Meters 2-Liters 3-Grams 4-End ";
120 AN$=INPUT$(1):AN=VAL (AN$)
130 IF AN<1 OR AN>4 THEN 120
140 PRINT AN$ 'Display response
150 ON AN GOTO 160,200,240,280
160 'Convert feet to meters
170 INPUT "Enter feet";FEET
180 PRINT FEET;"feet = ";FEET*.3048;"meters"
190 GOTO 110
200 'Convert gallons to liters
```



```

210 INPUT "Enter gallons";GAL
220 PRINT GAL;"gallons = ";GAL*3.785;"liters"
230 GOTO 110
240 'Convert ounces to grams
250 INPUT "Enter ounces";OUNCES
260 PRINT OUNCES;"ounces = ";OUNCES*28.349;"grams"
270 GOTO 110
280 'End execution
290 END

```

This program illustrates a common application of the **ON GOTO** and **ON GOSUB** statements. The program begins by displaying a list, or *menu*, of conversion options. To select one of the conversions, you enter the number of that option. Since the **INPUT\$** function is used to scan the keyboard, you do not need to press **ENTER** after your selection. After converting your response to a numeric value and checking that you have selected a valid option number, the program uses an **ON GOTO** statement to direct execution to the correct conversion routine. The purpose of the **PRINT** statement in line 140 is to display your response and cancel the pending print condition.

A sample program **RUN** appears below.

```

RUN
1-Meters 2-Liters 3-Grams 4-End 1
Enter feet? 45
  45 feet = 13.716 meters
1-Meters 2-Liters 3-Grams 4-End 3
Enter ounces? 89.9
  89.9 ounces = 2548.5751 grams
1-Meters 2-Liters 3-Grams 4-End 2
Enter gallons? 13.1
  13.1 gallons = 49.5835 liters
1-Meters 2-Liters 3-Grams 4-End 4
Ok

```

The next example illustrates an application of the **ON GOSUB** statement. This program calculates the day of the week for any date after October 15, 1582. (The program is not accurate for dates

before October 15, 1582, the year the Gregorian Calendar was instituted by Pope Gregory XIII.) The program uses a set of equations based on a formula known as Zeller's Congruence. The equations are:

$$1. T = 365 * \text{YEAR} + \text{DAY} + 31 * (\text{MO} - 1)$$

2. For January and February:

$$F = T + \text{INT}((\text{YEAR} - 1) / 4) - \text{INT}(.75 * (\text{INT}(((\text{YEAR} - 1) / 100)) + 1))$$

For March through December:

$$F = T - \text{INT}(.4 * \text{MO} + 2.3) + \text{INT}(\text{YEAR} / 4) - \text{INT}(.75 * (\text{INT}(\text{YEAR} / 100) + 1))$$

$$3. W = F + (-1 * (\text{INT}(F / 7) * 7))$$

where MO is the numeric month (1–12), DAY the calendar day (1–31), and YEAR is the year (1582+) of the target date. T and F are used for temporary storage of intermediate results. The final result of the calculation is a factor W (0–6) that indicates the day of the week based on the values shown below:

Value of W	Day of Week
0	Saturday
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday

Notice that Equation (2) is different depending on whether the month falls in the range January–February or March–December.

The program is given below.

```
NEW
100 'Day of week program
110 INPUT "Enter month";MO
120 INPUT "Enter day";DAY
```

```

130 INPUT "Enter year";YEAR
140 T=365*YEAR+DAY+31*(MO-1) 'Equation 1
150 IF MO=2 THEN GOSUB 200 ELSE GOSUB 220
    'Equation 2
160 W=F+(-1*(INT(F/7)*7)) 'Equation 3
170 ON W+1 GOSUB 240,250,260,270,280,290,300
180 PRINT "The day is ";D$
190 GOTO 110 'Loop
200 'Equation 2 for Jan-Feb
210 F=T+INT((YEAR-1)/4)-INT(.75*(INT(((YEAR-1)/
    100))+1)):RETURN
220 'Equation 2 for Mar-Dec
230 F=T-INT(.4*MO+2.3)+INT(YEAR/4)-
    INT(.75*(INT(YEAR/100)+1)):RETURN
240 D$="Saturday":RETURN
250 D$="Sunday":RETURN
260 D$="Monday":RETURN
270 D$="Tuesday":RETURN
280 D$="Wednesday":RETURN
290 D$="Thursday":RETURN
300 D$="Friday":RETURN

```

The **ON GOSUB** statement is used to assign the correct day of the week to D\$ after W has been calculated. (To keep the example short, no attempt is made to check that a valid date is entered. As a practice exercise, combine this program with the date-checking program given in Lesson 33 to ensure that only valid dates are entered.)

A sample **RUN** is shown below.

```

RUN
Enter month? 11
Enter day? 19
Enter year? 1946
The day is: Tuesday
Enter month? 4
Enter day? 25

```

```
Enter year? 2001
The day is: Wednesday
Enter month? 10
Enter day? 24
Enter year? 1929
The day is: Thursday
```

(Press **CTRL** **C** to halt program execution.)

SUMMARY

The **ON GOTO** and **ON GOSUB** statements are decision-making statements that differ from the **IF THEN** statement in that

- They base decisions on a numeric value and not on the results of a true-or-false test.
- They allow a single statement to transfer program execution to more than one program location.

The **ON GOTO** and **ON GOSUB** statements have the general form

```
ON numeric expression GOTO line list
ON numeric expression GOSUB line list
```

where *numeric expression* is a numeric variable or calculation and *line list* is a list of line numbers separated by commas.

The computer uses a simple counting scheme to select the line number in the *line list*. The program branches to the first line number if the value of *numeric expression* is 1, to the second line number if it is 2, to the third line number if it is 3, and so on. If *numeric expression* has a negative value, a ?FC Error condition occurs. If *numeric expression* has a value that is greater than the number of transfer locations in the *line list* or a value of 0, the **ON GOTO** or **ON GOSUB** statement is skipped.

LEFT\$, RIGHT\$, SPACE\$, and INSTR

One of the strongest features of the BASIC programming language is its many instructions for manipulating strings. This lesson concludes the survey of BASIC string functions with a discussion of the **LEFT\$**, **RIGHT\$**, **SPACE\$**, and **INSTR** functions.

The **LEFT\$** and **RIGHT\$** functions perform similar but opposite operations. As suggested by their names, **LEFT\$** returns the left portion of a string and **RIGHT\$** returns the right portion of a string.

The **LEFT\$** function has the general form

LEFT\$(string expression,length)

The value entered for *length* determines the number of characters returned by the function, counting from the left side of the string. If the value entered for *length* exceeds the length of *string expression*, **LEFT\$** returns a string equal to *string expression*.

The **RIGHT\$** function has the general form

RIGHT\$(string expression,length)

The value entered for *length* determines the number of characters returned by the function, counting from the right side of the string. If the value entered for *length* exceeds the length of *string expression*, **RIGHT\$** returns a string equal to *string expression*.

The following examples demonstrate the operation of these two functions.

```
T$="123456789"
Ok
PRINT LEFT$(T$,3)
123
Ok
PRINT RIGHT$(T$,3)
789
Ok
FOR C=1 TO 9:PRINT RIGHT$(T$,C):NEXT C
9
89
789
6789
56789
456789
3456789
23456789
123456789
Ok
```

The **SPACE\$** function creates a string of character spaces, up to a maximum of 255 spaces. The function has the general form

SPACE\$(length)

where *length* determines the number of spaces returned by the function.

The **LEFT\$**, **RIGHT\$**, and **SPACE\$** functions perform operations that can be accomplished with other BASIC instructions, although not as simply. For example, **MID\$** with the correct parameters can return the left or right portion of a string, and **STRING\$** can create a string of spaces. But the function discussed next cannot be duplicated by any other BASIC instruction, and is one of the most powerful and useful string functions.

The **INSTR** function compares two strings on a character-by-character basis, searching for a match in the first string of the characters in the second string. If a match is found, the function returns the position of the matching character(s). If no match is found, the function returns a 0.

The **INSTR** function has the general form

INSTR(*start position*,*search string*,*match string*)

The function compares the *search string* with the *match string*, starting at the character position identified by *start position*. The *start position* parameter is optional; if omitted, the **INSTR** function begins the search with the first character of the *search string*.

The following program demonstrates the **INSTR** function.

```
NEW
100 'INSTR example
110 A$="etaonrishdlfcmugypwbvkxjqz"
120 INPUT "Letter";L$
130 IF L$="" THEN 120
140 P=INSTR(1,A$,L$)
150 PRINT "Position =";P
160 GOTO 120
```

This program begins by assigning the lowercase letters of the alphabet to A\$. Then, after prompting you to enter a letter, the program uses **INSTR** to find the position of the letter in A\$. Line 150 displays the position and line 160 loops back for another entry.

A sample **RUN** is shown below.

```

RUN
Letter? a
Position = 3
Letter? u
Position = 15
Letter? *
Position = 0

```

(Press **CTRL** **C** to halt program execution.)

Note that A\$ is searched from left to right.

The letters in A\$ are not randomly arranged, as you might think, but are in order of usage in the English language. Thus, "e" is the most commonly used English letter and "z" is the least commonly used letter. If you enter a character that is not found in A\$ (including uppercase letters), a 0 position is returned.

One of the idiosyncrasies of the **INSTR** function is that it returns a 1 if the *target string* is a null string. Of course, if the first character in the *search string* matches the *target string*, the function will also return a 1. To prevent this quirk from introducing errors into a program, it is good practice to check for a null *target string* before performing the **INSTR** function. Line 130 in the preceding example performs this action.

The **INSTR** function is often combined with the **INPUT\$** or **INKEY\$** functions to create specialized keyboard input subroutines. A common application is a routine for selecting options from a menu of choices, as illustrated by the following example.

```

NEW
100 'Specialized input example
110 CLS
120 PRINT "<A> First Option"
130 PRINT "<B> Second Option"
140 PRINT "<C> Third Option"
150 PRINT:PRINT "Your Selection: ";
160 A$=INPUT$(1)
170 D=INSTR(1,"AaBbCc",A$)

```



```

180 ON D GOTO 600,600,700,700,800,800
190 BEEP:GOTO 160
600 PRINT "Option A selected":GOTO 120
700 PRINT "Option B selected":GOTO 120
800 PRINT "Option C selected":GOTO 120

```

The important parts of this program are lines 180 and 190. These lines determine whether an uppercase or lowercase A, B, or C has been pressed and, if so, transfer execution to the correct line number. (If this were a complete program rather than a demonstration example, lines 600, 700, and 800 would begin the sections that perform the menu options.)

The keyboard input routine illustrated by this program has at least three advantages: only valid keypresses are accepted by the program, correct responses are not restricted to a single letter case, and the program user is not required to press **ENTER** to execute a menu option.

This technique can be easily adapted to other special input situations. Simply change the value of the *target string* in the **INSTR** functions and add or subtract line numbers from the *line list* in the **ON GOTO** statement.

For an example of another specialized input routine, refer to Lesson 36.

SUMMARY

The **LEFT\$** and **RIGHT\$** functions perform similar but opposite operations: **LEFT\$** returns the left portion of a string and **RIGHT\$** returns the right portion of a string. The functions have the general form

```

LEFT$(string expression,length)
RIGHT$(string expression,length)

```

where *length* determines the number of characters returned by the function. **LEFT\$** counts from the left side and **RIGHT\$** counts from the right side of *string expression*. If the value entered for *length*

exceeds the length of *string expression*, the functions return a string equal to *string expression*.

The **SPACE\$** function creates a string of spaces, up to a maximum of 255 spaces. The function has the general form

SPACE\$(*length*)

where *length* determines the number of spaces returned by the function.

The **INSTR** function compares two strings on a character-by-character basis, searching for a match in the first string of the characters in the second string. If a match is found, the function returns the position of the matching character(s). If no match is found, the function returns a 0.

The **INSTR** function has the general form

INSTR(*start position*,*search string*,*match string*)

The *start position* parameter is optional; if omitted, the function begins the search with the first (leftmost) character of the *search string*. The *search string* is the string on which the search is performed; the *match string* identifies the character(s) searched for.

REVIEW TEST 7

1. Which of the following are valid BASIC statements?

- (a) 100 CLEAR 0
- (b) 340 IF INKEY\$="E" THEN END
- (c) 110 IF A<7 AND >14 THEN READ A\$
- (d) 400 IF XOR COST>100 THEN MARKUP=.15
- (e) 700 LINEINPUT "Zip Code: ";ZIP\$
- (f) 110 IF N\$="END" AND (B=5 OR C=1) THEN 120
- (g) 220 ON N\$ GOTO 10,70,170
- (h) 300 ON SGN(X)+2 GOTO 100,200,300
- (i) 310 ON PRICE<100 GOSUB 200 ELSE 250
- (j) 660 CLEAR FRE(A\$)
- (k) 500 AN\$=LINE INPUT\$(9)
- (l) 256 IF CC=40 OR RIGHT\$(3,T\$)="ity" THEN 200
- (m) 100 P\$=INKEY\$(1):GOTO 2001
- (n) 950 DEFINT A-Z:CLEAR 1000
- (o) 200 ON INSTR("1234",I\$) GOTO 210,220,230,210
- (p) 501 IF NOT NOT NOT A=5 THEN 9000
- (q) 155 A\$=INPUT\$(LEN("PASS"))
- (r) 886 TT\$=LEFT\$(L\$,4)+SPACES\$(12)+
RIGHT\$(R\$,4)

2. The **STR\$** function adds a leading space when it converts a positive number to a string value. Write a program that “strips” this space from strings converted from positive numbers but does not alter strings converted from negative numbers.
3. What is the purpose of the **CLEAR** statement. When is it used?
4. The computer’s display screen is 40 characters wide. Write a program that permits you to input a string value less than 40 characters in length, and then centers the value in the middle of a 40-character string of spaces.
5. Construct one-line programs to perform the following tasks.
 - (a) Display the first character of a string.
 - (b) Display all but the first character of a string.
 - (c) Display the last and next-to-last character of a string.
 - (d) Display the first and last characters of a string.
 - (e) Test whether a string contains a backslash (\).
 - (f) Display the first “e,” if any, in a string.
 - (g) Display the message “Press any key to continue” and halt program execution until a key is pressed.
6. Write a program that inputs names in last-name, first-name order, with a comma separating the two parts of the name, and then removes the comma and displays the names in first-name, last-name order.
7. Write a program that inputs string data and then replaces any S’s in the data with \$ signs.

Controlling the PRINT Position

The display screens of the Model 100 and PC-8201A consist of eight 40-character lines, for a total of 320 different character positions. The display screen of the Model 200 has sixteen 40-character lines, for a total of 640 character positions. By using the **PRINT @** statement on the Models 200 and 100, or the **LOCATE** statement on the PC-8201A, you can instruct the computer to begin **PRINTing** at any of these locations. This provides an alternative to the "scrolling" of **PRINTed** information that has been used thus far.

On the Models 200 and 100, the character positions are numbered consecutively, beginning with the upper left corner, as illustrated in Figs. 36-1 and 36-2. The first character position is 0; the last position is 319 on the Model 100 and 639 on the Model 200. On the PC-8201A, each character position is identified by a column-and-row number, as illustrated in Fig. 36-3. The top row is row 0; the leftmost column in each row is column 0. Thus, on the PC-8201A, the upper left position is column 0, row 0 and the lower right position is column 39, row 7.

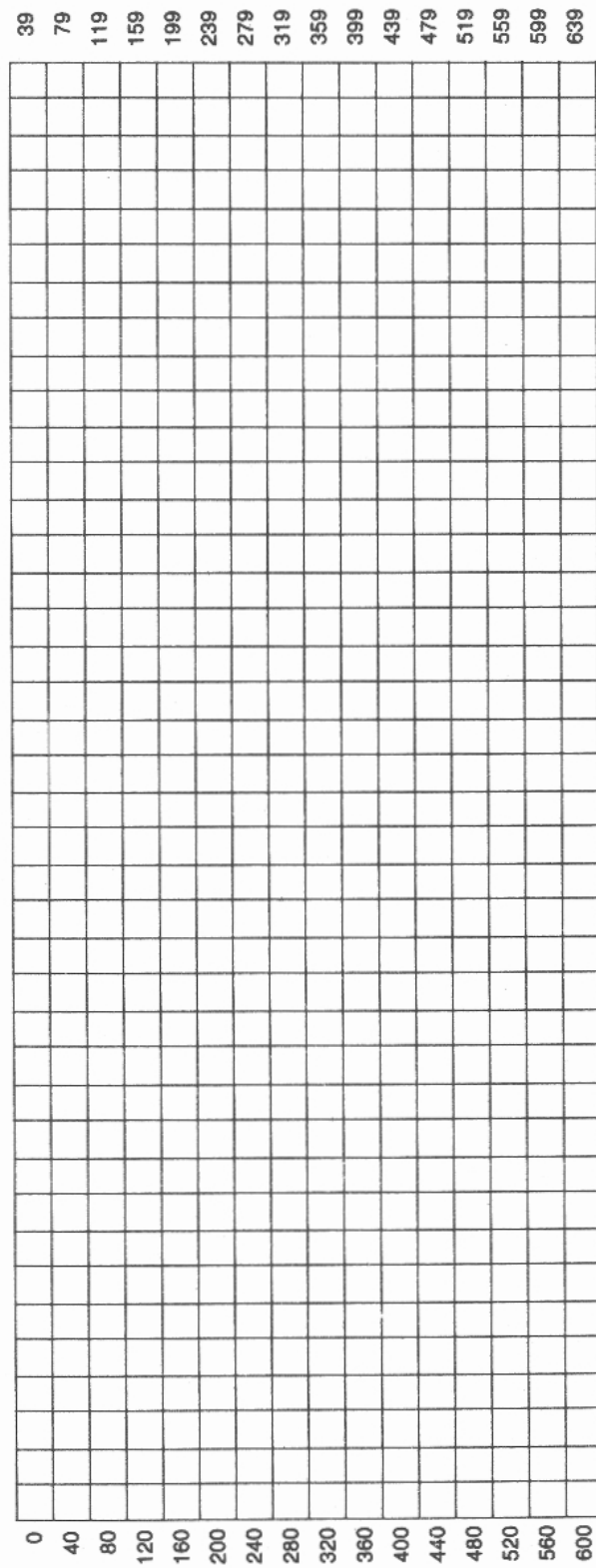


Fig. 36-1 Illustration showing the **PRINT @** values used by the Model 200.

The Models 200 and 100 use the @ (pronounced "at") option with the **PRINT** statement to instruct the computer to begin **PRINTing** at a specific screen location. The general form of the **PRINT** statement with the @ option is

PRINT @ *position,expression list*

where *position* identifies the screen location where **PRINTing** begins. A comma must separate the *position* parameter from the *expression list*.

The PC-8201A uses the **LOCATE** statement to control the **PRINT** position. The **LOCATE** statement positions the cursor to a desired column-and-row location. The next **PRINT** statement begins displaying its information at that location. The general form of the **LOCATE** statement is

LOCATE *column,row*

where *column* ranges from 0 to 39 and *row* ranges from 0 to 7. The following program demonstrates the screen-positioning capability of the computer. Two versions of the program are given, one for the Models 200 and 100 and one for the PC-8201A. Enter the version that is appropriate for your computer.

```
NEW
100 'Model 100/200 Version
110 CLS
120 FOR X=1 TO 1000
130 PRINT @ 138,X      (Replace 138 with 297 for Model 200.)
140 NEXT X

100 'PC-8201A Version
110 CLS
120 FOR X=1 TO 1000
130 LOCATE 18,4:PRINT X
140 NEXT X
```

When you **RUN** this program, the computer counts from 1 to 1000 and displays the result in approximately the middle of the screen.

Using the **PRINT @** or **LOCATE** statement to position printed information does not cause the display to “scroll” except when printing on the last line of the screen. To prevent scrolling when printing on the last line, follow the printed information with a semicolon, as illustrated in the next example.

```

NEW
100 'Model 100/200 Version
110 CLS
120 FOR X=1 TO 20
130 PRINT @ 290,X      (Replace 290 with 618 for Model 200.)
140 NEXT X
150 FOR X=1 TO 200
160 PRINT @ 290,X;     (Replace 290 with 618 for Model 200.)
170 NEXT X

100 'PC-8201A Version
110 CLS
120 FOR X=1 TO 20
130 LOCATE 10,7:PRINT X
140 NEXT X
150 FOR X=1 TO 200
160 LOCATE 10,7:PRINT X;
170 NEXT X

```

The only important difference in the two loops is the trailing semicolon in the second loop. The trailing semicolon prevents the scrolling action of the computer. Note also the vast difference in processing time used by the two loops. The first loop runs much slower because of the additional time required to scroll the entire display.

The next program provides another example of the power of **PRINT** positioning. It also illustrates a peculiarity of the computer—that it is impossible to prevent scrolling when a character is printed in the lower right corner of the display.

```

NEW
100 'Model 100/200 screen editor
110 SCREEN 0,0:CLS
120 CP=0 'Cursor Position

```

```

130 PRINT @ CP,"";
140 A$=INPUT$(1)
150 ON INSTR(1,CHR$(28)+CHR$(29)+CHR$(30)+CHR$(31),A$) GOSUB 180,190,200,210
160 IF ASC(A$)<32 OR ASC(A$)>122 THEN 130
170 PRINT A$;GOSUB 180:GOTO 130
180 IF CP+1>319 THEN RETURN ELSE
    CP=CP+1:RETURN      (Replace 319 with 639 for Model 200.)
190 IF CP-1<0 THEN RETURN ELSE CP=CP-1:RETURN
200 IF CP-40<0 THEN RETURN ELSE
    CP=CP-40:RETURN
210 IF CP+40>319 THEN RETURN ELSE
    CP=CP+40:RETURN      (Replace 319 with 639 for Model 200.)
100 'PC-8201A screen editor
110 SCREEN 0,0:CLS
120 C=0:R=0 'Column and Row position
130 LOCATE C,R
140 A$=INPUT$(1)
150 ON INSTR(1,CHR$(28)+CHR$(29)+CHR$(30)+CHR$(31),A$) GOSUB 180,200,220,230
160 IF ASC(A$)<32 OR ASC(A$)>122 THEN 130
170 PRINT A$;GOSUB 180:GOTO 130

180 IF C=39 AND R=7 THEN RETURN
190 IF C=39 THEN R=R+1:C=0:RETURN ELSE
    C=C+1:RETURN
200 IF C=0 AND R=0 THEN RETURN
210 IF C=0 THEN R=R-1:C=39:RETURN ELSE
    C=C-1:RETURN
220 IF R=0 THEN RETURN ELSE R=R-1:RETURN
230 IF R=7 THEN RETURN ELSE R=R+1:RETURN

```

This program creates a simple screen editor. The arrow keys move the cursor around the screen, and the typewriter keys display letters and symbols. The program demonstrates the complete control possible with the **PRINT @** or **LOCATE** statement. As an experiment move the cursor to the lower right corner and type a character—

the screen scrolls. This is the only screen location where a trailing semicolon cannot prevent scrolling.

A description of the program is given below.

Model 100/200 Version

- Line 110 Turns off the function key definitions and clears the screen.
- Line 120 Sets CP, the variable used for tracking the position of the cursor, to an initial value of 0.
- Line 130 Sets the **PRINT** position by **PRINT**ing a null string at location CP.
- Line 140 Accepts a single character from the keyboard. **INPUT\$** is used instead of **INKEY\$**, because the former displays a flashing cursor while waiting for input.
- Line 150 Uses the instruction sequence discussed in Lesson 35 to check for a cursor key. If  is pressed, the subroutine at line 180 is executed; if  is pressed, the subroutine at 190 is executed; if  is pressed, the subroutine at line 200 is executed; if  is pressed, the subroutine at line 210 is executed. The ASCII value of these keys is 28, 29, 30, and 31, respectively.
- Line 160 Checks for keys with ASCII values less than 32 or greater than 122. If a key is in this group, execution goes to line 130 for another keypress (causing the keystroke to be ignored).
- Line 170 This line is executed only for keys that fail the test in line 160. The character associated with the key is displayed, the subroutine at line 180 is called to advance the cursor count by 1, and execution is sent to 130 for a new keypress.

- Line 180 This subroutine moves the cursor right. If the cursor is not already positioned at the lower right corner, the value of CP is increased by 1.
- Line 190 This subroutine moves the cursor left. If the cursor is not already at position 0, the value of CP is decreased by 1.
- Line 200 This subroutine moves the cursor up. If the cursor is not already on the top line, the value of CP is decreased by 40 (subtracting 40 gives the corresponding location on the line above).
- Line 210 This subroutine moves the cursor down. If the cursor is not already on the bottom line, the value of CP is increased by 40.

PC-8201A Version *(Comments are supplied only for lines that differ from the Model 100/200 Version.)*

- Line 120 Sets C and R, the variables used to track the column and row values of the cursor, to 0.
- Line 130 Sets the **PRINT** position using the **LOCATE** statement.
- Line 150 Uses the instruction sequence discussed in Lesson 35 to check for a cursor key. If  is pressed, the subroutine at line 180 is executed; if  is pressed, the subroutine at 200 is executed; if  is pressed, the subroutine at line 220 is executed; if  is pressed, the subroutine at line 230 is executed. The ASCII value of these keys is 28, 29, 30, and 31, respectively.
- Line 180 This subroutine moves the cursor right. If the cursor is already at column 39 of row 7, no action is taken. If the cursor is at column 39, the value of R is increased by 1 and C is set to 0; otherwise, the value of C is increased by 1.

- Line 200 This subroutine moves the cursor left. If the cursor is already at column 0 of row 0, no action is taken. If the cursor is at column 0, the value of R is decreased by 1 and C is set to 39; otherwise, the value of C is decreased by 1.
- Line 220 This subroutine moves the cursor up. If the cursor is not already on the top row, the value of R is decreased by 1.
- Line 230 This subroutine moves the cursor down. If the cursor is not already on the bottom row, the value of R is increased by 1.

As you might expect, the computer keeps constant track of the cursor position, even when no cursor is displayed. A program can check the location of the cursor by using the **POS** and **CSRLIN** functions.

The **POS** function returns the column position of the cursor. The function has the general form.

POS (*numeric expression*)

The **POS** function uses a “dummy” numeric argument—an argument is required, but the value of the argument does not affect the function. The column value returned by the function ranges from 0 (first position on a line) to 39 (last position).

The **CSRLIN** function returns the row position of the cursor. Unlike the **POS** function, the **CSRLIN** function does not use an argument (not even a dummy argument). The row value returned by the function ranges from 0 (top row) to 7 (last row) on the Model 100 and PC-8201A, and 0 to 15 on the Model 200.

Technical Note: The **CSRLIN** function returns a value of 6 (14 on Model 200) for the last line of the display if the function key definitions are being displayed on the screen. This is the same value returned by the function for the next-to-last line of the display. This idiosyncrasy can cause confusion in a program. As noted in

Lesson 9, the function key definition display can be turned off with the **SCREEN 0,0** instruction.

Because the PC-8201A uses column and row values to position the cursor, the values returned by **POS** and **CSRLIN** can be used "as is" on that computer. Model 200 and 100 owners, however, must convert **POS** and **CSRLIN** values to a **PRINT @** location. The following formula will perform this calculation:

$$CP = C + R * 40$$

where CP is the calculated **PRINT @** value, C is the column value returned by **POS**, and R is the row value returned by **CSRLIN**.

The **POS** and **CSRLIN** functions permit a program to save the position of the cursor for later use, as demonstrated by the following example. This program contains a specialized input routine that can be used any time a program needs a yes-or-no response to a prompt. The input routine uses the **POS** and **CSRLIN** functions to find the correct location for printing the response.

```

NEW
100 'Model 100/200 yes/no routine
110 PRINT "Is the entered data correct? (Y/N) ";
120 GOSUB 5000
130 IF AN$="Y" THEN 200 ELSE 300
200 PRINT:PRINT AN$:GOTO 110
300 PRINT:PRINT AN$:GOTO 110
5000 'Yes/No Routine
5010 AN$="":CP=POS(0)+CSRLIN*40
5020 PRINT @CP,"";A$=INPUT$(1)
5030 ON INSTR(1, "NnYy "+CHR$(13),A$) GOTO
      5050,5050,5060,5060,5070,5080
5040 BEEP:GOTO 5020
5050 PRINT @CP,"No ";AN$="N":GOTO 5020
5060 PRINT @CP,"Yes";AN$="Y":GOTO 5020
5070 IF AN$="" OR AN$="N" THEN 5060 ELSE 5050
5080 IF AN$="" THEN 5020 ELSE RETURN

```

```

100 'PC-8201A yes/no routine
110 PRINT "Is the entered data correct? (Y/N) ";
120 GOSUB 5000
130 IF AN$="Y" THEN 200 ELSE 300
200 PRINT:PRINT AN$:GOTO 110
300 PRINT:PRINT AN$:GOTO 110
5000 'Yes/No Routine
5010 AN$="":C=POS(0):R=CSRLIN
5020 LOCATE C,R:A$=INPUT$(1)
5030 ON INSTR(1,"NnYy "+CHR$(13),A$) GOTO
      5050,5050,5060,5060,5070,5080
5040 BEEP:GOTO 5020
5050 PRINT "No ";:AN$="N":GOTO 5020
5060 PRINT "Yes";:AN$="Y":GOTO 5020
5070 IF AN$="" OR AN$="N" THEN 5060 ELSE 5050
5080 IF AN$="" THEN 5020 ELSE RETURN

```

A major programming goal is to make programs easy to use—a characteristic known as “user-friendly.” The yes/no input routine used in this program has the following user-friendly features:

- The routine does not accept invalid keyboard entries, avoiding confusion and errors.
- The routine permits you to respond with uppercase or lowercase letters.
- Because typing letters can be awkward if you have taken your fingers off the keyboard, the routine provides an alternative—pressing the space bar. Pressing the space bar alternates between selecting yes or no.
- The routine makes it easy to correct mistypes, because a selection is not confirmed until the **ENTER** key is pressed. If **ENTER** is pressed before a selection is made, the keypress is ignored.

Run the program and experiment with pressing Y, y, N, n, the space bar, and **ENTER**. Because the program is intended to demonstrate the input routine, selecting yes or no only displays the value of AN\$ and repeats the entry prompt. In a completed program, each option would perform differently.

To see the advantage of the **POS** and **CRSLIN** functions, change the prompt in line 110. For example, try

```
110 PRINT "Continue? (Y/N) ";
```

The input subroutine automatically adjusts to the length of the prompt and displays the yes/no responses in the correct spot.

As you can see from examining lines 110 to 130, this subroutine is easy to use. Simply call the subroutine after printing the appropriate prompt, and test the value of AN\$ to determine the response. The routine does not permit execution to return until yes or no has been selected.

A line-by-line description of the subroutine is given below.

Model 100/200 Version

- Line 5010 Set AN\$ to null and saves the position of the cursor.
- Line 5020 Positions the cursor by printing a null string and accepts a single character from the keyboard.
- Line 5030 Uses the **INSTR** function to determine if one of the desired keys has been pressed, and the **ON GOTO** statement to select the line to execute.
- Line 5040 Sends execution to line 5020. This line is executed only for keys that fail the test in line 5030.
- Line 5050 This line is executed when no is selected. The line displays the word No, sets AN\$ to N, and sends execution to line 5020.
- Line 5060 This line is executed when yes is selected. The line displays the word Yes, sets AN\$ to Y, and sends execution to line 5020.

- Line 5070 This line is executed when the space bar is pressed. If AN\$ is null (this occurs only on the first press of the space bar) or N, execution is transferred to line 5060 to select yes. If AN\$ is Y, execution is transferred to line 5050 to select no.
- Line 5080 This line is executed when **ENTER** is pressed. If AN\$ is null (indicating that no choice has been made), execution is sent back to line 5020 for another keypress. Otherwise, a selection has been made, and execution returns from the subroutine.

PC-8201A Version (*Comments are supplied only for lines that differ from the Model 100/200 Version.*)

- Line 5010 Sets AN\$ to null and saves the column and row values of the cursor.
- Line 5020 Positions the cursor and accepts a single character from the keyboard.

SUMMARY

By using the **PRINT @** statement on the Models 200 and 100, or the **LOCATE** statement on the PC-8201A, you can instruct the computer to begin **PRINTing** at any screen location. Controlling the print position provides an alternative to the scrolling method of displaying information.

The general form of the **PRINT @** statement is

PRINT @ *position,expression list*

where *position* gives the screen location where the **PRINTing** of *expression list* is to begin. The **PRINT @** *positions* range from 0 (upper left corner) to 319 (lower right corner) on the Model 100 and 0 to 639 on the Model 200. Figures 36-1 and 36-2 provide a graphic representation of these positions.

The general form of the **LOCATE** statement is

LOCATE *column,row*

where *column* and *row* give the screen position where the next **PRINT** statement begins displaying. *Column* values range from 0 to 39 and *row* values range from 0 to 7. Figure 36-3 provides a graphic representation of the *row* and *column* positions used by the PC-8201A.

To prevent scrolling when displaying information on the last line, follow the information being printed with a semicolon. Scrolling cannot be prevented when printing in the lower right corner of the display.

The **POS** and **CSRLIN** functions provide a method of checking on the location of the cursor, even when no cursor is being displayed. **POS** returns the column position and **CSRLIN** the line position of the cursor.

The **POS** function has the general form

POS(*numeric expression*)

where *numeric expression* is a dummy argument that does not affect the value return by the function. The **POS** function returns a value from 0 to 39.

The **CSRLIN** function returns a value from 0 to 7 on the Model 100 and PC-8201A, and 0 to 15 on the Model 200. It does not have an argument. (The **CSRLIN** function returns a value of 6 (14 on Model 200) for the last line of the screen when the function key definitions are being displayed. To avoid possible ambiguities in identifying the last line of the display, turn off the function key definition display with the **SCREEN 0,0** command.)

The **PRINT @** position for the Models 200 and 100 can be calculated from the **POS** and **CSRLIN** values with the equation

$$CP = C + R * 40$$

where CP is the **PRINT @** value, C is the column value, and R is the row value.

The Graphic Statements

Your computer's BASIC supplies three graphic statements: **LINE**, **PSET**, and **PRESET**. The **LINE** statement draws a line between any two pixels, the **PSET** (Point **SET**) statement "lights" a pixel, and the **PRESET** (Point **RESET**) turns off a pixel. A *pixel* is any one of the liquid crystal "spots" that make up the display screen. (The Model 100 and PC-8201A screens have 15,360 pixels; the Model 200 screen has 30,720 pixels.) For graphic purposes, each pixel is identified by a column-and-row location. The column values range from 0 to 239. The row values range from 0 to 63 on the Model 100 and PC-8201A, and from 0 to 127 on the Model 200. Thus, the pixel in the upper left corner is column 0, row 0, and the pixel in the lower right corner is column 239, row 63, on the Model 100 and PC-8201A, and column 239, row 127, on the Model 200.

The **LINE** statement has the general form

LINE (*c1,r1*)—(*c2,r2*),*on/off*,*box/fill*

where *c1,r1* is the column-and-row location where the line begins and *c2,r2* is the column-and-row location where the line ends. The *c1,r1* parameter is optional; if omitted, the line begins where the last **LINE** statement ended. The *on/off* parameter is an optional numeric value that determines whether the pixels in the line are turned on ("lighted") or turned off ("erased"). If the *on/off* value is odd (or omitted), the pixels are turned on. If the *on/off* value is even, the pixels are turned off. The *on/off* value must be in the range 0 to 255.

The *box/fill* parameter is an optional parameter consisting of either the letter B or the letters BF. Using a B instructs the computer to draw a box rather than a line, with diagonally opposite corners at *c1,r1* and *c2,r2*. Using a BF instructs the computer to draw a box and then "fill" in all the pixels within the box. The *on/off* parameter controls whether the box is drawn or filled with pixels turned on or off. An *on/off* value is obligatory when the *box/fill* parameter is used.

The **LINE** statement is more difficult to define than to use, as illustrated by the following examples. (*For the Model 200, replace each occurrence of the number 63 with the number 127.*)

Statement	Action
LINE (0,0) — (239,63)	Draws a line from the upper left to the lower right corner (pixels turned on)
LINE (0,0) — (239,63),1,BF	Draws a box along the outside edge of the screen and fills in the box (pixels turned on)
LINE (239,0) — (239,63),1	Draws a line along the right edge of the screen (pixels turned on)

Statement

LINE (239,0) - (239,63),0

Action

Erases the line drawn by the previous example (pixels turned off)

Two program examples illustrating the power of the **LINE** statement follow. The first program creates random lines and sounds; the second generates a bar graph (histogram) from values that you enter.

NEW

100 'Random line example

110 DEFINT C,L,P,R

120 A=RND(-VAL(RIGHT\$(TIME\$,2))-1)

130 CLS:PRINT " ";

140 FOR C=1 TO 70

150 C2=INT(239*RND(1)+1)

160 R2=INT(63*RND(1)+1) *(Replace 63 with 127 for Model 200.)*

170 P=INT(16383*RND(1)+1)

180 L=INT(12*RND(1)+1)

190 LINE -(C2,R2),1

200 SOUND P,L

210 NEXT C

220 GOTO 130

NEW

100 'Bar graph example

110 DEFINT C,M,N,R: DIM V(33)

120 INPUT "Maximum value";MX

130 SF=MX/63 'Scaling factor *(Replace 63 with 127 for Model 200.)*

140 INPUT "Number of values";N

150 IF N<1 OR N>33 THEN 140

160 FOR C=1 TO N

170 INPUT "Value";V(C)

180 IF V(C)>MX THEN PRINT "Too large":GOTO 170

190 NEXT C

200 CLS:C1=4:C2=9

```

210 FOR C=1 TO N
220 R2=63-INT(V(C)/SF)      (Replace 63 in lines 220 and 230 with
                             127 for Model 200.)
230 LINE (C1,63)-(C2,R2),1,BF
240 C1=C1+7:C2=C2+7
250 NEXT C
260 GOTO 260

```

The **PSET** and **PRESET** statements enable a program to turn specific pixels on and off. The statements have the general form

```

PSET (c,r)
PRESET (c,r)

```

where *c* is a column value from 0 to 239 and *r* is a row value from 1 to 63 (127 on the Model 200). **PSET** turns the pixel on; **PRESET** turns the pixel off.

Technical Note: The PC-8201A permits you to add an *on/off* parameter to the **PSET** and **PRESET** statements. Using an even *on/off* parameter reverses the function of the **PSET** or **PRESET** statement. Using an odd *on/off* parameter leaves the function of the statements unaltered. For example, **PSET** (7,11),2 turns the designated pixel off; **PRESET** (7,11),2 turns the same pixel on.

A common use for these instructions is plotting. The program that follows provides a simple example by plotting a sine wave.

```

NEW
100 'Plotting example
110 CLS:DEFINT Y
120 FR=.158:AMP=23
130 LINE (0,31)-(239,31),1      (Replace 31 in lines 130 and 150
                                with the number 63 for Model 200.)

140 FOR Y=0 TO 239
150 X=31+AMP*SIN(Y*FR)
160 PSET (Y,X)
170 NEXT Y
180 GOTO 180

```

The program calculates values of X for values of Y. The variable FR controls the frequency of the sine wave (number of periods). The variable AMP controls the amplitude. A value of 31 (63) is added to the result to place the function in the middle of the screen.

By changing the values of FR and AMP, you can change the appearance of the wave. For example, try using a frequency of .3 or an amplitude of 15. To display the values of X and Y as they are calculated, insert the line

```
165 PRINT @600,Y;INT(X); 'Model 200 version
165 PRINT @280,Y;INT(X); 'Model 100 version
165 LOCATE 2,7:PRINT Y;INT(X); 'PC-8201A version
```

The following program provides a final example of the **PSET** function. The program draws a circle on the display screen.

```
NEW
100 'Draw circle
110 CLS:DEFINT A,C,R
120 C=120:R=32:RAD=27
130 FOR ANGLE=1 TO 45
140 C1=C+RAD*COS(ANGLE)
150 R1=R+RAD*SIN(ANGLE)
160 PSET (C1,R1)
170 NEXT ANGLE
180 GOTO 180
```

The C and R values are the column and row coordinates of the center of the circle. RAD is the radius of the circle. Altering the value of these variables will change the location and size of the circle.

SUMMARY

The computer's display screen consists of liquid crystal "spots." Each spot is referred to as a *pixel*.

The **LINE** statement draws a line between any two pixels, the **PSET** statement turns on (“lights”) a pixel, and the **PRESET** statement turns off (“erases”) a pixel.

The **LINE** statement has the general form

LINE (*c1,r1*)—(*c2,r2*),*on/off*,*box/fill*

where *c1,r1* is the column-and-row location where the line begins and *c2,r2* is the column-and-row location where the line ends. Omitting the *c1,r1* parameter (and the surrounding parentheses, but not the hyphen) instructs the computer to begin the line where the last **LINE** statement ended.

The *on/off* parameter is an optional numeric value that determines whether the pixels in the line are turned on or off. Using an odd *on/off* value or omitting the parameter turns the pixels on; using an even *on/off* value turns the pixels off. The *on/off* value must be in the range 0 to 255.

The *box/fill* parameter is an optional parameter consisting of either the letter B or the letters BF. B instructs the computer to draw a box with diagonally opposite corners at *c1,r1* and *c2,r2*. BF instructs the computer to “fill” in the pixels in the box. An *on/off* parameter is required when the *box/fill* parameter is used.

The **PSET** and **PRESET** statements turn specific pixels on and off. The two statements have the general form

PSET (*c,r*)

PRESET (*c,r*)

where *c* is a column value and *r* is a row value. (The PC-8201A allows the use of an optional *on/off* parameter with these instructions.)

Data Files—Part 1

BASIC provides two methods of storing information for access by a program: **DATA** statements and data *files*. **DATA** statements, as discussed already, store information within a program in standard program lines. Data files store information outside a program.

Data files provide a flexibility that **DATA** statements do not. Because data in a file is stored separate from a program, it can be accessed by more than one program. It can also be permanently changed by a program, an option not available with **DATA** statements. (A program can change information **READ** from **DATA** lines, but it cannot save the altered information in new **DATA** statements—only a programmer can do that.)

In programming, the cost of flexibility is usually complexity. In place of the single statement required to store information in a program, data files require an **OPEN** statement, a series of statements to read or write the information to the file, and a **CLOSE** statement.

To reduce this complexity to manageable proportions, this discussion of data files is divided into two lessons. This lesson defines the file-access statements. The next lesson provides step-by-step procedures for reading and writing data files, and examples illustrating their use. The BASIC instructions used to create and manipulate data files are summarized below. Each instruction is defined in detail in the pages that follow.

OPEN	Prepares the computer to create or use a data file by defining the file's characteristics
MAXFILES	Defines the maximum number of files that can be in use at one time
PRINT #	Writes information to a data file
PRINT # USING	Writes formatted information to a data file
INPUT #	Reads information from a data file
INPUT\$	Reads a fixed number of characters from a data file
LINE INPUT #	Reads an entire record from a data file
EOF	Provides a program with a means of testing whether the end of a data file has been reached
CLOSE	Terminates a program access to a file
KILL	Erases a file

The **OPEN** statement prepares the computer to access a data file by defining the characteristics of the file. The **OPEN** statement is used both to create a new file and to prepare the computer to read from or write to a file that already exists. A new file is automatically created when the computer does not find an existing file matching the parameters of the **OPEN** statement.

The **OPEN** statement has the general form

```
OPEN "device:filename" FOR mode AS file number
```

The **OPEN** parameters are explained below.

- *device:filename*—The *device:filename* parameter identifies the location (device) of the file and the name of the file. If

the *device:filename* is a string constant, it must be enclosed in quotation marks. The *device* names discussed in this lesson are RAM and CAS (other devices can be named but are not discussed in this book). The *filename* must begin with a letter and be six characters or less in length. If a RAM data file is being **OPENed**, the *filename* should have a .DO extension. (See Lesson 10 for a discussion of *filename* extensions.)

- *mode*—The *mode* parameter must be one of the following words: INPUT, OUTPUT, or APPEND. The *mode* parameter specifies whether a file will be read from (INPUT), written to (OUTPUT), or added to (APPEND).
- *file number*—Every data file must be assigned an identification number when it is **OPENed**. This number provides the BASIC statements that read from and write to the file with an abbreviated method of referring to the file. Each open file must have a different *file number*. The value of *file number* must be between 1 and the value of **MAXFILES**. (As an option, you can precede the *file number* parameter with a # sign.)

Examples:

```
100 OPEN "RAM:FILE1.DO" FOR OUTPUT AS 1
110 OPEN "CAS:ASSETS" FOR INPUT AS 2
```

The first statement opens a file named FILE1.DO in the computer's random access memory (RAM). The file is for output only and is assigned the file number 1. The second statement opens a file named ASSETS on cassette tape. The file is opened for input only and is assigned the file number 2.

The **OPEN** statement produces an error condition if:

- The characteristics of an existing file and those of the **OPEN** statement do not match.
- The *device* specified by the **OPEN** statement does not exist or is not connected to the computer.

- The *filename* exceeds six characters in length or contains invalid characters.
- An attempt is made to **OPEN** more files at one time than is permitted by the **MAXFILES** statement.
- An attempt is made to **OPEN** a file that is already **OPEN**.

The **MAXFILES** statement permits you to check or define the number of data files that can be **OPEN** at one time, up to a maximum of 15 files. When BASIC is selected initially, the number of files that can be active at one time is set to 1. If you need more than one file open at a time, use the **MAXFILES** statement to set a new limit.

Examples:

```
MAXFILES=4
PRINT MAXFILES
```

The first statement sets the number of files that can be **OPEN** at one time to 4. The second statement displays the current active file limit.

The **PRINT #** statement writes (stores) information to a data file that has been **OPENed** in OUTPUT or APPEND mode. Writing to a file opened in **INPUT** mode causes a ?BN Error.

The **PRINT #** statement has the general form

```
PRINT # file number,expression list
```

The *file number* parameter must match the number assigned to the file by the **OPEN** statement. The *expression list* identifies the information to be written to the file. If an *expression list* contains string values and you want to be able to read the values as separate items, you must write commas to the file with the data (referred to as *explicit* commas). The commas instruct the **INPUT #** statement (discussed next) to treat the string values as separate items. Numeric values do not have to be separated by explicit commas.

Example:

```
100 PRINT #1,T,"";T$,"";T#;T!
```

This statement writes the values of T, T\$, T#, and T! to file #1. Explicit commas are needed to separate the value of T\$ from T and T#.

The **PRINT # USING** statement instructs the computer to follow a specific format when writing data to a file. The format is defined by a *format string*, just as it is when displaying formatted information on the screen. The format characters available for defining the *format string* are discussed in Lesson 21.

The **PRINT # USING** statement has the general form

```
PRINT # file number USING format string; expression list
```

The information in the *expression list* is written to the file identified by the *file number*, using the format defined by the *format string*.

Example:

```
235 PRINT #1 USING "#####.##";PRICE
```

This statement writes the value of PRICE to file #1 using the format #####.##.

The **INPUT #** statement reads information from a data file that has been **OPENed** in **INPUT** mode. Reading from a file that has been opened in **OUTPUT** or **APPEND** mode causes a ?ID Error.

The **INPUT #** statement has the general form

```
INPUT # file number, variable list
```

The *file number* parameter must match the number assigned to the file by the **OPEN** statement. The *variable list* identifies the variables to be assigned the information read from the file. The variables in the list must be separated by commas.

Example:

```
300 INPUT #1,T,T$,T#,T!
```

This statement reads four data values from file #1 and assigns the information to the variables T, T\$, T#, and T!.

The **INPUT\$** function reads a fixed number of characters from a file. The function has the general form

INPUT\$(*numeric expression*,*file number*)

The *numeric expression* parameter defines the number of characters to read from the file identified by *file number*. (As an option, you can precede the *file number* parameter with a # sign.)

Example:

200 A\$=INPUT\$(19,2)

This statement reads 19 characters from file #2.

The **LINE INPUT #** statement reads an entire record from a data file, including any explicit commas in the record. If part of the record has been read already, the **LINE INPUT #** statement reads only the remainder of the record. A record can be a single data value or a series of data values, depending on how the file was created. The next lesson provides more details on what constitutes a record.

The **LINE INPUT #** statement has the general form

LINE INPUT # *file number*,*string variable*

The *file number* identifies the file to be read. The information read from the file is assigned to *string variable*.

Example:

335 LINE INPUT #1,R\$

This statement reads the next record from file #1 and assigns the information to R\$.

The **EOF** function tests whether additional information can be read from a data file. The **EOF** function has the general form

EOF (*file number*)

The *file number* parameter must match the number assigned to the file by the **OPEN** statement.

Example:

```
350 IF EOF(1) THEN CLOSE:END
```

This line tests the end-of-file condition of the file opened with the file number 1. If all the information in the file has been read already, the **CLOSE** and **END** statements are executed. Otherwise, the statements are skipped.

The **CLOSE** statement instructs the computer to terminate access to a data file. A closed file cannot be accessed again until another **OPEN** statement is performed.

The **CLOSE** statement has the general form

```
CLOSE file number, file number, . . .
```

The *file number(s)* specify which files to **CLOSE**. If the *file number* parameter is omitted, BASIC closes all open files. (As an option, you can precede the *file number* parameter with a # sign.)

It is vital to **CLOSE** an open data file. If a file is not properly closed, information in the file can be lost.

Example:

```
500 CLOSE 1,2
```

This line closes the files identified by the file numbers 1 and 2.

The **KILL** command erases a file from RAM (the command will not erase a file from cassette tape). Once a file is deleted, it cannot be recovered.

The **KILL** command has the general form

```
KILL "device:filename"
```

The **KILL** command requires that you specify the file extension along with the filename. The *device* parameter is optional, since the command can only delete a file from RAM.

Example:

KILL "DATES.DO"

This command erases the file named DATES.DO.

SUMMARY

The BASIC instructions used to create and manipulate data files are summarized below.

OPEN	Prepares the computer to create or use a data file by defining the file's characteristics
MAXFILES	Defines the maximum number of files that can be in use at one time
PRINT #	Writes information to a data file
PRINT # USING	Writes formatted information to a data file
INPUT #	Reads information from a data file
INPUT\$	Reads a fixed number of characters from a data file
LINE INPUT #	Reads an entire record from a data file
EOF	Provides a program with a means of testing whether the end of a data file has been reached
CLOSE	Terminates program access to a file
KILL	Erases a file

Data Files—Part 2

The preceding lesson defined the BASIC instructions available for creating and manipulating data files. This lesson provides step-by-step procedures for using those instructions to read, write, or add to a data file.

Computers traditionally permit the creation of two types of data files: *sequential* data files and *relative* data files. Since relative data files cannot be created in RAM or on cassette tape, this discussion is limited to sequential data files.

A sequential file can be visualized as a series of individual data values stored in a long row. This arrangement makes data access very simple—just start with the first data value and read or write until the last data value is reached. When a sequential data file is opened in OUTPUT mode, the computer is set to start writing at the beginning of the file, even if the file already contains data. If

a file matching the characteristics of the **OPEN** statement already exists, *any information previously stored in the file is lost.*

If you want to add information to the end of a sequential file, you can open the file in APPEND mode. APPEND mode instructs the computer to permit a program to write to the end of the file. If a new file is opened in APPEND mode, the computer is set to write to the beginning of the file, just as if the file had been opened in OUTPUT mode.

When a sequential file is opened in INPUT mode, the computer is set to read from the beginning of the file. There are no exceptions to this rule. Thus, to access a data value in the middle of a sequential file, it is necessary to read from the beginning of the file until you reach the item of information you are seeking.

To create a sequential data file:

1. **OPEN** the file in OUTPUT mode. Be aware that if you are opening an existing file in OUTPUT mode, the previous information in the file is lost.
2. Write the data to the file using the **PRINT #** statement, in the order that you want the information stored.
3. **CLOSE** the file when the last item of data has been written to the file.

The following program demonstrates the process of creating a sequential file in RAM.

```

NEW
100 'Sequential OUTPUT example
110 PRINT "Now opening the file. . ."
120 OPEN "RAM:OUT1.DO" FOR OUTPUT AS 1
130 PRINT "Now writing to the file. . ."
140 PRINT #1,"This is an example of a relatively long data
    item"
150 PRINT #1,"Now"
160 PRINT #1,"a"
170 PRINT #1,"series"
180 PRINT #1,"of"

```

```
190 PRINT #1,"shorter"  
200 PRINT #1,"data"  
210 PRINT #1,"items"  
220 PRINT "Now closing the file. . ."  
230 CLOSE 1  
240 PRINT "Done!"
```

This program creates the data file in RAM. The same data could be stored on cassette tape by changing RAM in the OPEN statement to CAS (of course, a cassette recorder must be connected to the computer and the RECORD button of the recorder pressed before the program is **RUN**).

RUN this program.

Because the file is created in RAM, it is possible to examine the file directly using the TEXT program. To do so, enter **MENU** to return to the Main Menu screen, direct the cursor to the name OUT1.DO and press **ENTER**. The computer will display the contents of the OUT1.DO file. Notice the left-pointed triangle following each line. This is the symbol for a carriage return (**ENTER** keypress) and marks the end of a data *record*. The length of a data record is determined by the *expression list* of the **PRINT #** statement. If the *expression list* contains one data item, the record will be one data value in length. If the *expression list* contains nine data items, the record will be nine data values in length.

Technical Note: If the *expression list* of a **PRINT #** statement ends in a semicolon, a pending print condition is established. The next **PRINT #** statement adds to these data values, without creating a new record. (This result is analogous to placing a semicolon after the *expression list* of a **PRINT** statement, and is used for similar reasons.)

To read a sequential file:

1. **OPEN** the file in INPUT mode. When you open a file in INPUT mode the computer is set to begin reading at the beginning of the file.

2. Read the data in the file using **INPUT #**, **INPUT\$**, or **LINE INPUT #** statements, depending on your needs. Use the **EOF** function before each file input statement to ensure that the program does not attempt to read past the end of the file.
3. **CLOSE** the file when the last item of data has been read (or all the information needed has been read).

The following program demonstrates the general procedure to read a sequential file. This program reads and displays the information written to the file named OUT1.DO by the preceding example.

```

NEW
100 'Sequential INPUT example
110 PRINT "Now opening the file. . ."
120 OPEN "RAM:OUT1.DO" FOR INPUT AS 1
130 PRINT "Now reading the file. . ."
140 'Test if end of file has been reached
150 IF EOF(1) THEN 190
160 INPUT #1,A$
170 PRINT A$
180 GOTO 150
190 PRINT "Now closing the file. . ."
200 CLOSE 1
210 PRINT "DONE!"

```

The major difference between the writing and reading procedures is in the use of the **EOF** function. The **EOF** function provides a program with a means of testing whether any records remain unread in a sequential file. If the **EOF** function is not used, the program will eventually attempt to input a data value when none remain to be read, causing a ?EF Error (End of File) condition.

Observe that the **EOF** function produces a true result only when the end of the file is reached.

In the preceding examples, each record contained only one string value. If you want to store more than one string item per record and have the strings readable as separate values, you must write

the data with commas between the individual data values. (The commas must be written to the file—not used as *expression list* separators.) For example,

```
200 PRINT #1,A$;" ";T$;" ";1234"
```

writes the values of A\$, T\$, and 1234 with commas separating the values. This permits a statement such as

```
800 INPUT #1,Z$,R$,S$
```

to read the record as three values.

When creating files containing both numeric and string data, you should know the order or pattern of the data in the file. Otherwise, you may find it difficult to read the data back into the computer successfully. The following program provides an example of storing numeric and string data in the same file; it also demonstrates adding data to an existing file using the APPEND mode.

The first part of the program enables you to select the type of file operation you want to perform from a menu of choices.

```
NEW
100 'Check tracking program
110 CLS:PRINT "1. Create a new file"
120 PRINT "2. Add to the file"
130 PRINT "3. Total the checks"
140 PRINT "4. Delete the file"
150 PRINT "5. End the program"
160 PRINT "Your selection? (1-5)";
170 A$=INPUT$(1):PRINT
180 ON INSTR(1,"12345",A$) GOTO 210,290,370,520,550
190 BEEP:GOTO 110
```

The next part of the program creates a new file and permits data to be stored in the file. If you press **ENTER** in response to the `Check Payable To` prompt, the program assumes that you have no more checks to enter and returns to the menu. (The file is closed before execution returns to the menu.)

```

200 'Create a new file
210 OPEN "RAM:CHECKS.DO" FOR OUTPUT AS 1
220 LINE INPUT "Check Payable To? ";PAYEE$
230 IF PAYEE$="" THEN CLOSE 1:GOTO 110
240 INPUT "Amount";AMOUNT
250 PRINT #1, PAYEE$
260 PRINT #1,AMOUNT
270 GOTO 220

```

Lines 280 through 350 add data to an existing file. Again, pressing **ENTER** in response to the Check Payable To prompt ends the entry of data.

```

280 'Add to an existing file
290 OPEN "RAM:CHECKS.DO" FOR APPEND AS 1
300 LINE INPUT "Check Payable To? ";PAYEE$
310 IF PAYEE$="" THEN CLOSE 1:GOTO 110
320 INPUT "Amount";AMOUNT
330 PRINT #1,PAYEE$
340 PRINT #1,AMOUNT
350 GOTO 300

```

Lines 360 through 500 read and display the file data, count the number of checks, and total the amounts. When the end of the file is reached, the number of checks and the total amount are displayed.

```

360 'Total the file
370 SUM=0:COUNT=0
380 OPEN "RAM:CHECKS.DO" FOR INPUT AS 1
390 IF EOF(1) THEN 450
400 LINE INPUT #1,PAYEE$
410 INPUT #1,AMOUNT
420 PRINT PAYEE$,AMOUNT
430 SUM=SUM+AMOUNT:COUNT=COUNT+1
440 GOTO 390
450 CLOSE 1
460 PRINT "Number of checks=";COUNT
470 PRINT "Total =";SUM
480 PRINT "Press any key to continue:";

```

```
490 A$=INPUT$(1)
500 GOTO 110
```

The next section erases the file from memory. Once a file is **KILLED**, the information in the file is lost.

```
510 'Erase the file
520 KILL "RAM:CHECKS.DO"
530 GOTO 110
```

The last lines end the program.

```
540 'End program
550 END
```

RUN the program and experiment with the various options. Note, however, if you attempt to total or delete the file before you store any data in the file, the program halts with an error condition. For instructions on how you can prevent these errors from stopping program execution, refer to the error-handling section of Lesson 40.

The program uses the **LINE INPUT** statement to enter the name of the payee so that you can use commas in the name, if desired. Since the data file may contain commas, the **LINE INPUT #** statement must be used to read the payee name. After creating a file, you may want to use the TEXT program to observe the storage format. In fact, as long as you follow the correct pattern, you can even store data in the file using TEXT.

To change this program to store data on cassette tape, change RAM to CAS in lines 210, 290, and 380. (You cannot **KILL** a file on cassette tape, so option 4 should not be used.)

SUMMARY

To create a sequential data file:

1. **OPEN** the file in OUTPUT mode. Be aware that if you are opening an existing file in OUTPUT mode, the previous information in the file is lost.

2. Write the data to the file using the **PRINT #** statement, in the order that you want the information stored.
3. **CLOSE** the file when the last item of data has been written to the file.

To add to a sequential data file:

1. **OPEN** the file in APPEND mode.
2. Write the additional data to the file using the **PRINT #** statement, in the order that you want the information added to the file.
3. **CLOSE** the file when the last item of data has been written.

To read a sequential file:

1. **OPEN** the file in INPUT mode. When you open a file in INPUT mode the computer is set to begin reading at the beginning of the file.
2. Read the data in the file using **INPUT #**, **INPUT\$**, or **LINE INPUT #** statements, depending on your needs. Use the **EOF** function before each file input statement to ensure that the program does not attempt to read past the end of the file.
3. **CLOSE** the file when the last item of data has been read (or all the information needed has been read).

40

273

- Programs saved in ASCII format can be manipulated as data files by file-access statements.

The command for saving a program in ASCII format has the general form

```
SAVE "device:filename",A
```

The following example demonstrates the process of "merging" two programs. Enter the following short program.

```
NEW
100 REM PROGRAM 1—LINE 100
110 REM PROGRAM 1—LINE 110
140 REM PROGRAM 1—LINE 140
150 REM PROGRAM 1—LINE 150
```

Now save the program in RAM using the A option.

```
SAVE "RAM:PGM1",A
```

Since the program is stored in ASCII format, the computer automatically adds a .DO extension to the filename (you can check this with the **FILES** command). Next clear the computer's memory. Then enter the following program. Notice that the first and second programs overlap at lines 110 and 140.

```
NEW
110 REM PROGRAM 2—LINE 110
120 REM PROGRAM 2—LINE 120
130 REM PROGRAM 2—LINE 130
140 REM PROGRAM 2—LINE 140
```

To **MERGE** the stored program with the program in memory, enter

```
MERGE "RAM:PGM1"
```

When the flashing cursor reappears, **LIST** the program. The program should appear as shown below.

LIST

```

100 REM PROGRAM 1—LINE 100
110 REM PROGRAM 1—LINE 110
120 REM PROGRAM 2—LINE 120
130 REM PROGRAM 2—LINE 130
140 REM PROGRAM 1—LINE 140
150 REM PROGRAM 1—LINE 150

```

The program saved as PGM1 has been merged with the program in the BASIC workspace. Where lines of the two programs overlapped, the lines from PGM1 have replaced the lines in the workspace. The lines that did not overlap have been inserted in the proper numeric order. Merging does not alter the stored program (PGM1).

TIME AND DATE

One of the major features of the computer is its capability to keep the time and date. The internal clock used by the computer is accurate to within several minutes a year, provided that battery power is not interrupted. The computer's clock is set from BASIC by assigning the appropriate values to **TIME\$**, **DATE\$**, and **DAYS**, as defined below.

TIME\$ The format for setting the time is: **TIME\$**="HH:MM:SS", where HH is the current hour in military (24-hour) format, MM is the current minutes, and SS is the current seconds. The HH:MM:SS value must be enclosed in quotes.

DATE\$ The date setting is maintained differently on the Models 200 and 100 and the PC-8201A. On the Models 200 and 100, the format for setting the date is **DATE\$**="MM/DD/YY". On the PC-8201A, the format for setting the date is **DATE\$**="YY/MM/DD". For setting the date, MM is the month, DD is the numerical day, and YY is the last two digits of the year. The date must be enclosed in quotes.

DAYS The format for setting the day of the week is **DAYS**="DDD", where DDD is one of the following

three-letter abbreviations: MON, TUE, WED, THU, FRI, SAT, or SUN. The day must be enclosed in quotes. (The **DAYS** instruction is not available on the PC-8201A.)

The current clock setting can be checked at any time by displaying the value of these variables. **TIMES** returns the current time in HH:MM:SS format, **DATES** returns the current date in MM/DD/YY format on the Models 200 and 100, and YY/MM/DD format on the PC-8201A, and **DAYS** returns the day of the week in DDD format.

The expression used in this book to get an unpredictable seed value for the pseudorandom number generator is an example of one use of these functions. The expression finds the current seconds setting of the clock by taking the **VAL** of the two rightmost characters returned by **TIMES**. Because the seconds setting can be 00, 1 is subtracted from the value to ensure that the seed number is never 0. (Using a 0 seed number instructs the computer to repeat the last pseudorandom number.) The resulting value is negated to obtain a negative seed value.

PRINTER INSTRUCTIONS

BASIC supplies five instructions for directly addressing a printer: **LCOPY**, **LLIST**, **LPRINT**, **LPRINT USING**, and **LPOS**. The action of these statements is defined below.

- **LCOPY**—Prints the current text contents of the display screen. (This instruction is available only on the Models 200 and 100.)
- **LLIST** *line number range*—Prints a program listing. **LLIST** can be used with or without a *line number range* parameter. If used without the *line number range*, the computer prints the entire program; otherwise, it prints only the lines named in the *line number range*.
- **LPRINT** *expression list*—Prints the value of *expression list*. **LPRINT** acts identically to the **PRINT** statement, except the value of *expression list* is sent to the printer instead of to the display screen.

- **LPRINT USING** *"format string";expression list*—Prints *expression list* using the format specified by *format string*. The rules that define the *format string* are described in Lesson 21.
- **LPOS** (*numeric expression*)—Returns the current column position of the print head of the printer. **LPOS** uses a "dummy" argument that does not alter the value returned by the function.

ERROR HANDLING

The error-handling instructions of BASIC enable a program to avoid halting execution and displaying an error message when an error occurs. With these instructions, a program can determine which error has occurred and take appropriate action, without human intervention. The error-handling instructions are **ON ERROR GOTO**, **RESUME**, **ERL**, **ERR**, and **ERROR**.

The **ON ERROR GOTO** statement defines an error-handling location by instructing the computer to transfer program execution to a specific line number if an error occurs. The statement has the general form

ON ERROR GOTO *line number*

where *line number* determines the line to execute when an error occurs. If the value of *line number* is 0, any previous **ON ERROR GOTO** statements are canceled and the computer returns to its normal mode of halting when an error occurs.

Technical Note: Executing a **CLEAR** or **MAXFILES** statement cancels an **ON ERROR GOTO** statement.

The **RESUME** statement is used in an error-handling routine to instruct the computer to cancel an error and resume program execution. The statement has two general forms:

RESUME NEXT
RESUME *line number*

If **NEXT** is used, program execution resumes with the first line following the line where the error occurred. If a *line number* other than 0 is specified, execution resumes at the specified program line. If a 0 *line number* is specified, execution resumes at the same line where the error occurred.

The **ERL** function returns the line number where the last error occurred. If the error occurred in command mode, the function returns 65535.

The **ERR** function returns the error code of the last error. The error codes used by the computer are given in Appendix C.

The **ERROR** statement simulates the occurrence of an error. The advantage of this statement is that it makes possible to check error-handling routines without having to wait until an error occurs. The **ERROR** statement has the general form

ERROR *error code*

where *error code* identifies the error to simulate.

The following program demonstrates the capabilities of the error-handling instructions.

```

NEW
100 'Error handling example
110 ON ERROR GOTO 500
120 READ A$
130 PRINT A$
140 GOTO 120
500 IF ERL=120 AND ERR=4 THEN END
510 PRINT "UNEXPECTED ERROR";ERR;"in line";ERL:END
800 DATA ineluctable,modality, laua
810 DATA hyrax,perdure,ergonomics
820 DATA quark,egocidal,fianchetto
830 DATA sacerdotal,tope,caryatids

```

As you know from the discussion in Lesson 18, attempting to read more data items than exist in a program causes an ?OD Error (Out of Data). The usual method of preventing this (besides counting

the data items) is to place dummy values at the end of the list and test for their occurrence. In the sample program shown above, an error-handling routine is used to determine when the last data item has been read. When an error occurs, the routine beginning at line 500 is executed. If an error 4 is found to have occurred in line 120, the last data item has been read and program execution is ended. If any other error occurs, the message UNEXPECTED ERROR, the code assigned to the error, and line number in which the error occurred is displayed.

Error code 4 is assigned to the ?OD Error. Although you can find the code assigned to any error by looking it up in Appendix C, you can also use the **ERROR** statement. For example, entering

```
ERROR 4
```

displays the message

```
?OD Error
```

It is a good idea when designing an error-handling routine to include code for unexpected errors. In the preceding example, line 510 performs this function. You can test the effectiveness of this line by introducing another error. For example, enter

```
125 ERROR 2
```

and run the program again. This simulated syntax error causes the program to display the message:

```
UNEXPECTED ERROR 2 in line 125
```

Perhaps the most common application for error-handling routines is in programs that use data files. Perhaps, because of inadequacies in the file-access statements, errors are very common when manipulating files. The lines shown below are designed to be added to the check tracking program developed in Lesson 39. Besides making the program more user friendly, the lines illustrate a practical error-handling routine.

```

105 ON ERROR GOTO 600
600 'Error routine for checks program
610 BEEP
620 IF ERR=57 AND ERL=210 THEN PRINT "CANNOT
    OPEN CHECKS FILE—TOO MANY FILES":GOTO 670
630 IF ERR=57 AND ERL=290 THEN PRINT "CANNOT
    ADD TO CHECKS FILE—TOO MANY FILES":GOTO 670
640 IF ERR=52 AND ERL=380 THEN PRINT "CANNOT
    TOTAL FILE—FILE DOES NOT EXIST":GOTO 670
650 IF ERR=52 AND ERL=520 THEN PRINT "CANNOT
    DELETE FILE—FILE DOES NOT EXIST":GOTO 670
660 PRINT "UNEXPECTED ERROR";ERR;"in line"; ERL
670 CLOSE
680 FOR C=1 TO 1500:NEXT 'Delay
690 RESUME 110

```

OPTIMIZING PROGRAMS FOR SPACE AND SPEED

In your day-to-day programming efforts, you will often find that you must reduce the size of program if it is going to fit into memory, or that you must speed up the program to make its use acceptable. Be warned, however, that fixing a program to take less memory or execute quicker usually makes it more difficult to understand or debug.

The following hints will help you reduce the memory requirements of a program.

- Remove all unnecessary spaces. For example, changing

```
100 IF A=5 THEN PRINT "DONE"
```

to

```
100 IFA=5THENPRINT"DONE"
```

saves 4 bytes.
- Remove all or most **REMARKs**.
- Use as many multiple-statement lines as possible.
- Delete all unnecessary lines.

- Use the variable type declaration statements to keep variable storage to a minimum.
- Dimension arrays to the exact size needed.
- **CLEAR** only as much string space as required by the program.

The following hints will help you speed up a program.

- Drop the *control variable* following the **NEXT** statement in **FOR NEXT** loops. (This parameter is optional.) For example, 100 FOR C=1 TO 100:NEXT will execute faster than 100 FOR C=1 TO 100:NEXT C. Dropping the *control variable* not only speeds up the program, it saves program space.
- Use integers where possible, especially for **FOR NEXT** loops.
- Place subroutines at the beginning of a program. (BASIC searches for a subroutine starting at the first line of a program, so placing subroutines near the beginning reduces this search time.)
- When possible, design your programs so that a minimum of instructions are placed in loops. If a loop executes 500 times, each instruction within the loop also executes 500 times. If rearranging a program reduces the number of instructions within a loop, the time savings can be dramatic.
- If a constant is used more than once in a program, assign it to a variable and use the variable instead of the constant. Converting a constant to the format used by a program is more time consuming than using the value of a variable.

REVIEW TEST 8

1. Which of the following are valid BASIC statements?

- (a) 100 C=POS:R=CSRLIN
- (b) 340 PRINT @40,USING " # # # # #.#";M
- (c) 886 A\$=ERROR 2
- (d) 110 LOCATE 250,5
- (e) 345 INPUT #1,DT\$:IF EOF(1) THEN CLOSE
- (f) 400 LINE (CSRLIN,POS(9))-(0,0),1
- (g) 200 ON ERROR GOSUB 5000
- (h) 220 PO=PRESET(33,11)
- (i) 300 F=MAXFILES+1
- (j) 310 OPEN A\$ FOR INPUT AS E
- (k) 660 PRINT #2,USINGF\$;EL\$
- (l) 500 ZZ\$=INPUT\$(123,4)
- (m) 256 CLOSE #2
- (n) 100 SAVE"CAS:IOTA.DO",A
- (o) 110 LINE -(7,11),255,BF
- (p) 950 A\$=LCOPY
- (q) 501 ON ERROR GOTO 100,200,200
- (r) 155 RESTORE NEXT
- (s) 999 E1=ERR:E2=ERL:E3=EOF

- (t) 123 MERGE"CAS:" + F\$ + ".DO"
- (u) 700 LINE (0,1) - (250,60),B
- (v) 567 OPEN DAY\$ FOR OUTPUT AS 1

2. Explain the purpose of a trailing semicolon after a **PRINT @** statement on the Models 200 and 100 or after a **PRINT** statement following a **LOCATE** statement on the PC-8201A.
3. Modify the screen editor program given in Lesson 36 so that it displays a tic-tac-toe board and restricts the characters that you can type to Xs and Os.
4. Write a program that lights every other pixel on the display screen.
5. What is a data file? What is a file record? What kind of information can be stored in a data file?
6. What special technique must be used to separate strings stored in the same record so that they can later be read as individual values?
7. Where must new information be added to a sequential data file?
8. Can a program change the name of a data file?
9. Your computer's BASIC does not provide a command for making a copy of a program. Write a program that performs this function. Design the program so that it prompts for the name of the file to copy and the filename to assign to the copy.
10. Write a program that allows you to input the following information and stores the data in a RAM file. Design the program so that

it averages the scores of each student and stores that average as the last data value of the student's record.

	<u>Test 1</u>	<u>Test 2</u>	<u>Test 3</u>	<u>Average</u>
Student 1	78	82	89	?
Student 2	83	85	81	?
Student 3	100	70	73	?
Student 4	65	68	71	?
Student 5	99	89	93	?
Student 6	87	0	93	?

APPENDIX A

ASCII Codes and Characters

Table I lists the ASCII codes used by the Tandy Models 200 and 100, and the NEC PC-8201A. The decimal and hexadecimal values of the codes are given in the columns titled Dec and Hex. The string character (if any) displayed by the codes is given in the column titled Character. The key used to generate the codes is shown in the column titled Key Sequence.

TABLE I

ASCII Code		Model 100/200 Key		PC-8201 Key	
Dec	Hex	Character	Sequence	Character	Sequence
00	00		PAUSE		PAST
01	01		CTRL A		CTRL A
02	02		CTRL B		CTRL B
03	03		CTRL C		CTRL C

(continued)

TABLE I (Continued)

ASCII Code		Model 100/200		PC-8201A	
Dec	Hex	Character	Key Sequence	Character	Key Sequence
04	04	(BEEP)	D		D
05	05		E		E
06	06		F		F
07	07		G		G
08	08		H/		H/
09	09		I/		I/
10	0A		J		J
11	0B		K		K
12	0C		L		L
13	0D		M/		M/
14	0E		N		N
15	0F		O		O
16	10		P		P
17	11		Q		Q
18	12		R		R/
19	13		S		S
20	14		T		T
21	15		U		U
33	16		V		V
23	17		W		W
24	18		X		X
25	19		Y		Y
26	1A		Z		Z
27	1B				
28	1C				
29	1D				
30	1E				
31	1F				
32	20				

TABLE I (Continued)

ASCII Code		Model 100/200		PC-8201A	
Dec	Hex	Character	Key Sequence	Character	Key Sequence
33	21	!	!	!	!
34	22	"	"	"	"
35	23	#	#	#	#
36	24	\$	\$	\$	\$
37	25	%	%	%	%
38	26	&	&	&	&
39	27	,	,	,	,
40	28	(	(	(	(
41	29	)	)	)	)
42	2A	*	*	*	*
43	2B	+	+	+	+
44	2C	,	,	,	,
45	2D	—	—	—	—
46	2E	.	.	.	.
47	2F	/	/	/	/
48	30	0	0	0	0
49	31	1	1	1	1
50	32	2	2	2	2
51	33	3	3	3	3
52	34	4	4	4	4
53	35	5	5	5	5
54	36	6	6	6	6
55	37	7	7	7	7
56	38	8	8	8	8
57	39	9	9	9	9
58	3A	:	:	:	:
59	3B	;	;	;	;
60	3C	<	<	<	<
61	3D	=	=	=	=
62	3E	>	>	>	>
63	3F	?	?	?	?

(continued)

TABLE I (Continued)

ASCII Code		Model 100/200 Key		PC-8201A Key	
Dec	Hex	Character	Sequence	Character	Sequence
64	40	@	@	@	@
65	41	A	A	A	A
66	42	B	B	B	B
67	43	C	C	C	C
68	44	D	D	D	D
69	45	E	E	E	E
70	46	F	F	F	F
71	47	G	G	G	G
72	48	H	H	H	H
73	49	I	I	I	I
74	4A	J	J	J	J
75	4B	K	K	K	K
76	4C	L	L	L	L
77	4D	M	M	M	M
78	4E	N	N	N	N
79	4F	O	O	O	O
80	50	P	P	P	P
81	51	Q	Q	Q	Q
82	52	R	R	R	R
83	53	S	S	S	S
84	54	T	T	T	T
85	55	U	U	U	U
86	56	V	V	V	V
87	57	W	W	W	W
88	58	X	X	X	X
89	59	Y	Y	Y	Y
90	5A	Z	Z	Z	Z
91	5B	[	[	[	[
92	5C	\	GRAPH —	\	\
93	5D	]	]	]	]
94	5E	~	~	~	~

TABLE I (Continued)

ASCII Code		Model 100/200		PC-8201A	
Dec	Hex	Character	Key Sequence	Character	Key Sequence
95	5F	—	—	—	—
96	60	,	GRAPH [	,	—
97	61	a	a	a	a
98	62	b	b	b	b
99	63	c	c	c	c
100	64	d	d	d	d
101	65	e	e	e	e
102	66	f	f	f	f
103	67	g	g	g	g
104	68	h	h	h	h
105	69	i	i	i	i
106	6A	j	j	j	j
107	6B	k	k	k	k
108	6C	l	l	l	l
109	6D	m	m	m	m
110	6E	n	n	n	n
111	6F	o	o	o	o
112	70	p	p	p	p
113	71	q	q	q	q
115	72	r	r	r	r
115	73	s	s	s	s
116	74	t	t	t	t
117	75	u	u	u	u
118	76	v	v	v	v
119	77	w	w	w	w
120	78	x	x	x	x
121	79	y	y	y	y
122	7A	z	z	z	z
123	7B	{	GRAPH 9	{	{
124	7C	:	GRAPH SHIFT —	:	:
125	7D	}	GRAPH 0	}	}

(continued)

TABLE I (Continued)

ASCII Code		Model 100/200 Key		PC-8201A Key
Dec	Hex	Character	Sequence	Character Sequence
126	7E	~	GRAPH SHIFT [	~
127	7F		DEL	DEL
128	80		GRAPH P	
129	81		GRAPH M	
130	82		GRAPH F	
131	83		GRAPH X	
132	84		GRAPH C	
133	85		GRAPH A	
134	86		GRAPH H	
135	87		GRAPH T	
136	88	i	GRAPH 1	
137	89		GRAPH R	
138	8A	$\neq$	GRAPH /	
139	8B	Σ	GRAPH S	
140	8C	$\approx$	GRAPH '	
141	8D	$\pm$	GRAPH =	
142	8E	$\int$	GRAPH I	
143	8F		GRAPH E	
144	90		GRAPH Y	
145	91		GRAPH U	
146	92		GRAPH ;	
147	93		GRAPH Q	
148	94		GRAPH W	
149	95	σ	GRAPH B	
150	96	ϕ	GRAPH N	
151	97	$\%$	GRAPH .	
152	98		GRAPH O	
153	99		GRAPH ,	
154	9A		GRAPH L	
155	9B		GRAPH K	
156	9C		GRAPH 2	

TABLE I (Continued)

ASCII Code		Model 100/200		PC-8201	
Dec	Hex	Character	Key Sequence	Character	Key Sequence
157	9D	◊	GRAPH 3		
158	9E	♥	GRAPH 4		
159	9F	◊	GRAPH 5		
160	A0	·	CODE ,		
161	A1	à	CODE X		
162	A2	ç	CODE C		
163	A3	£	CODE 8		
164	A4	·	CODE SHIFT ,		
165	A5	μ	CODE SHIFT M		
166	A6	°	CODE SHIFT)		
167	A7	▼	CODE SHIFT —		
168	A8	†	CODE SHIFT +		
169	A9	§	CODE S		
170	AA	█	CODE SHIFT R		
171	AB	█	CODE SHIFT C		
172	AC	¼	CODE P		
173	AD	¾	CODE ;		
174	AE	½	CODE /		
175	AF	¶	CODE 0		
176	B0	¥	GRAPH 7		
177	B1	Ä	CODE SHIFT A		
178	B2	Ö	CODE SHIFT O		
179	B3	Ü	CODE SHIFT U		
180	B4	φ	CODE 6		
181	B5	~	CODE [		
182	B6	ä	CODE A		
183	B7	ö	CODE O		
184	B8	ü	CODE U		
185	B9	B	CODE SHIFT S		
186	BA	™	CODE SHIFT T		
187	BB	é	CODE D		

(continued)

TABLE I (Continued)

ASCII Code		Model 100/200 Key		PC-8201 Key	
Dec	Hex	Character	Sequence	Character	Sequence
188	BC	ù	CODE ,		
189	BD	è	CODE V		
190	BE	..	CODE =		
191	BF	f	CODE SHIFT F		
192	C0	â	CODE 1		
193	C1	ê	CODE 3		
194	C2	î	CODE 8		
195	C3	ô	CODE 9		
196	C4	û	CODE 7		
197	C5	^^	CODE —		
198	C6	ë	CODE E		
199	C7	ï	CODE I		
200	C8	á	CODE Q		
201	C9	í	CODE K		
202	CA	ó	CODE L		
203	CB	ú	CODE J		
204	CC	ý	CODE Y		
205	CD	ñ	CODE N		
206	CE	ã	CODE Z		
207	CF	õ	CODE .		
208	D0	Â	CODE SHIFT !		
209	D1	Ê	CODE SHIFT #		
210	D2	Î	CODE SHIFT *		
211	D3	Ô	CODE SHIFT (		
212	D4	Û	CODE SHIFT &		
213	D5	Ï	CODE SHIFT I		
214	D6	Ë	CODE SHIFT E		
215	D7	É	CODE SHIFT D		
216	D8	Á	CODE SHIFT Q		
217	D9	Í	CODE SHIFT K		
218	DA	Ó	CODE SHIFT L		

TABLE I (Continued)

ASCII Code		Model 100/200		PC-8201	
Dec	Hex	Character	Key Sequence	Character	Key Sequence
219	DB	Ú	CODE SHIFT J		
220	DC	Ý	CODE SHIFT Y		
221	DD	Ù	CODE SHIFT <		
222	DE	È	CODE SHIFT V		
223	DF	À	CODE SHIFT X		
224	E0		GRAPH SHIFT Z		
225	E1	■ (upper left)	GRAPH SHIFT 1		
226	E2	■ (upper right)	GRAPH SHIFT 2		
227	E3	■ (lower left)	GRAPH SHIFT 3		
228	E4	■ (lower right)	GRAPH SHIFT 4		
229	E5	■	GRAPH SHIFT 5		
230	E6	■	GRAPH SHIFT 6		
231	E7	— (upper)	GRAPH SHIFT Q		
232	E8	— (lower)	GRAPH SHIFT W		
233	E9	(left)	GRAPH SHIFT E		
234	EA	(right)	GRAPH SHIFT R		
235	EB	┐	GRAPH SHIFT A		
236	EC	└	GRAPH SHIFT S		
237	ED	┌	GRAPH SHIFT D		
238	EE	└	GRAPH SHIFT F		
239	EF	■	GRAPH SHIFT X		
240	F0	ˆ	GRAPH SHIFT U		
241	F1	—	GRAPH SHIFT P		
242	F2	ˆ	GRAPH SHIFT O		
243	F3	ˆ	GRAPH SHIFT I		
244	F4	ˆ	GRAPH SHIFT J		
245	F5		GRAPH SHIFT :		

(continued)

TABLE I (Continued)

ASCII Code		Model 100/200 Key			PC-8201 Key	
Dec	Hex	Character	Sequence			Character Sequence
246	F6	⌞	GRAPH	SHIFT	M	
247	F7	⌟	GRAPH	SHIFT	.	
248	F8	⌠	GRAPH	SHIFT	,	
249	F9	⌡	GRAPH	SHIFT	L	
250	FA	⌢	GRAPH	SHIFT	K	
251	FB	⌣	GRAPH	SHIFT	H	
252	FC	⌤	GRAPH	SHIFT	T	
253	FD	⌥	GRAPH	SHIFT	G	
254	FE	⌦	GRAPH	SHIFT	Y	
255	FF	⌧	GRAPH	SHIFT	C	

Table II lists the ASCII sequences assigned special functions by the Tandy Models 200 and 100, and the NEC PC-8201A. Because these sequences begin with the ASCII code assigned to the **ESC** key, they are often referred to as "escape" sequences.

TABLE II

ASCII Sequence	Action
27 65	Moves the cursor up one row†
27 66	Moves the cursor down one row†
27 67	Moves the cursor right one column†
27 68	Moves the cursor left one column†
27 69	Clears the screen and positions the cursor to column 0, row 0
27 72	Homes the cursor
27 74	Erases from the cursor position to the end of the screen
27 75	Erases from the cursor position to the end of the row
27 76	Inserts a row of spaces at the cursor position and moves the cursor to the first column of the row
27 77	Deletes the row at the cursor position, moves the following rows up one line, and moves the cursor to the first column of the row
27 80	Turns off the cursor
27 81	Turns on the cursor
27 84	Locks the bottom row of the screen so that it does not scroll with the rest of the display (any information PRINT ed on the bottom line prior to executing this sequence is preserved)
27 85	Cancels the effect of the preceding sequence
27 86	Locks the bottom row of the screen so that when information is PRINT ed on the bottom line, it

(continued)

† These functions do not "wrap around." Thus, **PRINT**ing CHR\$(27)+CHR\$(67) in the last column of a row does not advance the cursor to the first column of the next row.

TABLE II (*Continued*)

ASCII Sequence	Action
	does not cause the display to scroll
27 87	Cancels the effect of the preceding sequence
27 89	Positions the cursor to a specific row and column location as explained later in this appendix
27 106	Clears the screen and positions the cursor to column 0, row 0
27 108	Erases the current row without changing the position of the cursor
27 112	Places the display in reverse video mode
27 113	Cancels reverse video mode

The functions given in Table II are executed by **PRINT**ing the characters represented by the designated ASCII values (using the **CHR\$** function). The following program demonstrates several of the functions. (*For the Model 200, replace the number 292 in line 120 with the number 600.*)

```

100 'Model 100/200 version
110 SCREEN 0,0
120 PRINT @292, "PROTECTED LINE";
130 PRINT CHR$(27)+CHR$(84) 'Lock last row
140 FOR C=10 to 30
150 PRINT C;STRING$(31,42);C
160 NEXT C
170 PRINT @120,"WILL DELETE THIS LINE . . .";
180 FOR C=1 to 2000:NEXT C
190 PRINT @120,CHR$(27)+CHR$(77) 'Delete row
200 FOR C=1 TO 1000:NEXT C
210 PRINT @40, "ERASE FROM CURSOR . . .";

```



```

220 FOR C=1 TO 2000:NEXT C
230 PRINT CHR$(27)+CHR$(74) 'Erase from cursor
240 PRINT CHR$(27)+CHR$(85) 'Unlock last row
250 GOTO 250

100 'PC-8201A version
110 SCREEN 0,0
120 LOCATE 12,7:PRINT "PROTECTED LINE";
130 PRINT CHR$(27)+CHR$(84) 'Lock last row
140 FOR C=10 TO 30
150 PRINT C;STRING$(31,42);C
160 NEXT C
170 LOCATE 0,3:PRINT "WILL DELETE THIS LINE . . .";
180 FOR C=1 TO 2000:NEXT C
190 LOCATE 0,3:PRINT CHR$(27)+CHR$(77) 'Delete row
200 FOR C=1 TO 1000:NEXT C
210 LOCATE 0,1:PRINT "ERASE FROM CURSOR . . .";
220 FOR C=1 TO 2000:NEXT C
230 PRINT CHR$(27)+CHR$(74) 'Erase from cursor
240 PRINT CHR$(27)+CHR$(85) 'Unlock last row
250 GOTO 250

```

As noted in the table, the ASCII values 27 89 can be used to position the cursor to any row-and-column location. The format for this sequence is

$\text{CHR}\$(27) + \text{CHR}\$(89) + \text{CHR}\$(\text{row value}) + \text{CHR}\(column value)

The ASCII values used to identify the row and column positions are shown in Table 3. For example, the sequence

$\text{PRINT CHR}\$(27) + \text{CHR}\$(89) + \text{CHR}\$(37) + \text{CHR}\(49)

instructs the computer to position the cursor at row 5, column 17.

TABLE III

ASCII Value	Row/Column Position	ASCII Value	Row/Column Position
32	0	52	20
33	1	53	21
34	2	54	22
35	3	55	23
36	4	56	24
37	5	57	25
38	6	58	26
39	7	59	27
40	8	60	28
41	9	61	29
42	10	62	30
43	11	63	31
44	12	64	32
45	13	65	33
46	14	66	34
47	15	67	35
48	16	68	36
49	17	69	37
50	18	70	38
51	19	71	39

APPENDIX B

BASIC Reserved Words

This appendix provides an alphabetic listing of the BASIC reserved words. A reserved word cannot be used as a variable name or as part of a variable name. Words that are reserved only on the Models 200 and 100 are marked with an asterisk (*). Words that are reserved only on the PC-8201A are marked with a plus(+).

ABS	CINT	CRSLIN
AND	CLEAR	DATA
ASC	CLOAD	DATE\$
ATN	CLOADM *	DAY\$ *
BEEP	CLOSE	DEFDBL
BLOAD +	CLS	DEFINT
BLOAD? +	COM	DEFSNG
BSAVE +	CONT	DEFSTR
CALL *	COS	DELETE
CDBL	CSAVE	DIM
CHR\$	CSNG	EDIT

ELSE	LLIST	PSET
END	LOAD	READ
EOF	LOADM *	REM
EQV	LOC	RENUM +
ERL	LOCATE +	RESTORE
ERR	LOF	RESUME
ERROR	LOG	RETURN
EXEC +	LPOS	RIGHT\$
EXP	LPRINT	RND
FILES	MAXFILES	RUN
FIX	MAXRAM	RUNM *
FOR	MDM *	SAVE
FRE	MENU	SAVEM *
GOSUB	MERGE	SCREEN
GOTO	MID\$	SGN
HIMEM	MOD	SIN
IF	MOTOR	SOUND
IMP	NAME	SPACE\$
INKEY	NEW	SQR
INP	NEXT	STEP
INPUT	NOT	STOP
INSTR	OFF	STR\$
INT	ON	STRING\$
IPL *	OPEN	TAB
KEY	OR	TAN
KILL	OUT	THEN
LCOPY *	PEEK	TIMES\$
LEFT\$	POKE *	TO
LEN	POS	USING
LET	POWER	VAL
LINE	PRESET	VARPTR *
LIST	PRINT	XOR

APPENDIX C

Error Codes and Messages

The following table lists the error codes and messages assigned to the Tandy Models 200 and 100.

Code	Message	Meaning
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	RETURN with GOSUB
4	OD	Out of Data
5	FC	Illegal Function Call
6	OV	Overflow
7	OM	Out of Memory
8	UL	Undefined Line
9	BS	Bad Subscript
10	DD	Doubly Dimensioned array
11	/0	Division by zero

Code	Message	Meaning
12	ID	Illegal Direct
13	TM	Type Mismatch
14	OS	Out of String space
15	LS	String too Long
16	ST	String formula Too complex
17	CN	Can't continue
18	IO	IO error
19	NR	No RESUME
20	RW	RESUME Without error
21	UE	Undefined Error
22	MO	Missing Operand
23-49	UE	Undefined Error
50	IE	Internal Error
51	BN	Bad file Number
52	FF	File not Found
53	AO	File Already Open
54	EF	Input past End of File
55	NM	Bad file Name
56	DS	Direct Statement in file
57	FL	File Limit (RAM directory full)
58	CF	File not open
59-255	UE	Undefined Error

The following table lists the error codes and messages assigned to the NEC PC-8201A.

Code	Message	Meaning
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	RETURN with GOSUB
4	OD	Out of Data
5	FC	Illegal Function Call
6	OV	Overflow
7	OM	Out of Memory
8	UL	Undefined Line
9	BS	Bad Subscript

Code	Message	Meaning
10	DD	Doubly Dimensioned array
11	/0	Division by zero
12	ID	Illegal Direct
13	TM	Type Mismatch
14	OS	Out of String space
15	LS	String too Long
16	ST	String formula Too complex
17	ON	Can't continue
18	UF	Undefined Function
19	NR	No RESUME
20	RW	RESUME Without error
21	UE	Undefined Error
22	MO	Missing Operand
23	BO	Keyboard Buffer Overflow
24	IO	IO error
25	DU	Device Unavailable
26-49	UE	Undefined Error
50	IE	Internal Error
51	BN	Bad file Number
52	FF	File not Found
53	AO	File Already Open
54	EF	Input past End of File
55	NM	Bad file Name
56	DS	Direct Statement in file
57	FL	File Limit (RAM directory full)
58	CF	File not open
59	PC	PC-8001 error
60-255	UE	Undefined Error

APPENDIX D

Answers to Review Tests

REVIEW TEST 1

1. The **PRINT** 1,,3 statement is a legal BASIC sequence that places 1 in the first print zone and 3 in the third print zone. The second print zone is left blank.
2. The comma separator instructs the computer to display the next print item in the next print zone. The semicolon instructs the computer to display the next item immediately following the preceding item.
3. (a) Valid (the extra spaces do not affect the operation of the statement)
(b) Valid
(c) Valid (using seven commas instructs the computer to display the 7 in the seventh display zone)
(d) Valid
(e) Invalid (**PRINT** is not a valid variable name)

4. The initial value of a numeric variable is zero; the initial value of a string variable is null.
5.
 - (a) Valid (" " is the null string)
 - (b) Valid
 - (c) Valid (the computer will interpret 100 as a line number, not as part of a variable name)
 - (d) Invalid (cannot assign values to a constant)
 - (e) Valid (the **LET** is optional)
 - (f) Valid
 - (g) Valid
 - (h) Invalid (the variable must be on the left side of the = sign)
 - (i) Valid
 - (j) Invalid (**NEW** is on the reserved word list)
6.
 - (a) Valid
 - (b) Valid
 - (c) Invalid (period not allowed in a variable name)
 - (d) Valid
 - (e) Valid
 - (f) Valid (BASIC is not a reserved word)
 - (g) Invalid (@ sign not allowed in a variable name)
 - (h) Valid
 - (i) Invalid (hyphen not allowed in a variable name)
 - (j) Invalid (TREND contains the reserved word **END**)
 - (k) Valid
 - (l) Invalid (variable names cannot begin with a digit; if you enter a sequence such as 1st=5, BASIC will interpret the 1 as a line number)
7. Variable names must begin with a letter of the alphabet, only the first two characters of the name are significant, and they cannot contain words on the reserved word list. String variables must end with a \$ sign.
8. Yes.

9. To insert a new line, assign it a line number between the existing line numbers where you want it to appear. If a program is numbered with no gaps between the lines, you cannot insert a new line unless you change the line numbers. The **EDIT** command can be used to alter line numbers.
10. To replace a line, enter the new line with the same line number as the old.
11. Use the **NEW** command to erase an entire program from the BASIC workspace memory. To erase a single line, enter just the line number.
12. The **EDIT** command enables you to edit the program currently in the BASIC workspace. The command permits you to edit the entire program, or only a single line or range of lines. The commands available for editing when using the built-in TEXT program are also available for editing the program.

REVIEW TEST 2

1. (a) Invalid (the prompt is not enclosed in quotes)
(b) Invalid (a comma is used to separate multiple input variables—not a semicolon)
(c) Invalid (this example mixes two types of assignment statements: the **INPUT** statement, which assigns keyboard input to a variable, and the assignment sequence, which assigns a value to a variable)
(d) Invalid (the **INPUT** statement accepts only a constant for a prompt)
(e) Valid
(f) Invalid (a semicolon must separate a prompt and an input variable)

2. A program that counts by 9s in character positions 1 and 15 is shown below.

```
100 PRINT X,  
110 X=X+9  
120 GOTO 100
```

A program that counts backward from 2475 by 99 and displays the values adjacently is shown below.

```
100 X=2475  
110 PRINT X;  
120 X=X-99  
130 GOTO 110
```

3. A *pending display state* instructs the computer to preserve the present contents of the display when the next **PRINT** statement is executed. It is created by placing a semicolon or comma after the *expression list* of a **PRINT** statement. The advantage of the pending display state is that it allows information to be jointly displayed by different **PRINT** statements.
4. The completed program is given below.

```
100 INPUT "Current balance";BALANCE  
110 INPUT "Amount of check or deposit";AMOUNT  
120 BALANCE=BALANCE+AMOUNT  
130 PRINT "New balance = $";BALANCE  
140 GOTO 110
```

A sample **RUN** of the program is shown below.

```
RUN  
Current balance? 127.18  
Amount of check or deposit? -29.95  
New balance = $ 97.23  
Amount of check or deposit? -4.51  
New balance = $ 92.72  
Amount of check or deposit? -67.12  
New balance = $ 25.6
```

```

Amount of check or deposit? -49.95
New balance = $-24.35
Amount of check or deposit? 300
New balance = $ 275.65

```

(Press **CTRL** **C** to halt program execution.)

5. A "counting variable" is commonly used to control the number of times a loop is executed. The counting variable is set to some initial value before the loop is entered and is increased or decreased by one each time the loop is executed. An **IF THEN** test is used to determine when a specified termination value has been reached. When the termination value is reached, execution is transferred out of the loop.
6. (a) Valid (although valid, notice that the test $X=X+1$ can never have a true result. An **IF THEN** tests the truth or falsehood of a condition; it does not perform an assignment operation)
 (b) Valid (true and false are valid variable names)
 (c) Valid (the not-equal-to sign can be entered as $<>$ or $><$; the former is the customary way to enter it)
 (d) Invalid (this type of relational expression is not valid in BASIC)
 (e) Valid
 (f) Valid (the **GOTO** is optional)
 (g) Valid (an **IF THEN** statement is valid as the action of another **IF THEN** statement)
7. A program that solves this problem is shown below.

```

100 INPUT "Number of sets ordered";N
110 C=N*14.88
120 IF N=>5000 THEN DISCOUNT=C*.4
130 IF N<5000 THEN DISCOUNT=C*.35
140 IF N<=1000 THEN DISCOUNT=C*.3
150 IF N<=500 THEN DISCOUNT=C*.25
160 IF N<=100 THEN DISCOUNT=C*.2

```

```

170 PRICE=C-DISCOUNT
180 PRINT "Wholesale price = $";PRICE
190 GOTO 100

```

A sample **RUN** appears below.

```

RUN
Number of sets ordered? 433
Wholesale price = $ 4832.28
Number of sets ordered? 888
Wholesale price = $ 9249.408
Number of sets ordered? 2001
Wholesale price = $ 19353.672

```

(Press **CTRL** **C** to halt program execution.)

8. A file *extension* is a code added after a filename to provide a method of grouping files into categories. The extensions discussed in this book are .BA (for BASIC files) and .DO (for TEXT files). The **NAME** and **KILL** commands require that you specify the extension with a filename.
9. (a) Assigns the command **FILES** to function key 1.
 (b) Renames the file CARDS.BA to DRAW.BA.
 (c) Erases the file named MENU.BA.
 (d) Saves the program currently in the BASIC workspace. The value of F\$ determines the file's name. If the value of F\$ is not a valid filename, a ?NM Error occurs.
 (e) Loads the file named ASSETS.BA into the BASIC workspace.
 (f) Displays a listing of the RAM files.
 (g) Loads and executes the file named SOUND.BA.

REVIEW TEST 3

1. The phrase "levels of precedence" refers to the ranking system used by the computer to determine the order in which mathemat-

ical operations are performed. The function of parentheses in a calculation is to force the computer to evaluate an operation or series of operations in an order different from that resulting from the levels of precedence.

2. The square root of a number can be calculated by raising the value to the $\frac{1}{2}$ power. The fifth root can be calculated by raising the value to the $\frac{1}{5}$ power.
3. The unary minus operation will be performed first.
4. (a) 100 PRINT SQR(12+77)/3.14159
 (b) 100 PRINT 118.56*2.022^(2+3.8)
 (c) 100 PRINT 1/9+1/7+1/5+1/3 (No parentheses required.)
 (d) 100 PRINT (142/117+93/12)/(16/40) (Parentheses are needed to force the division operations to be performed in the correct order.)
 (e) 100 PRINT (SIN(206)-TAN(94))/COS(171)
5. The "argument" of a function is the value used by the function to perform its operation. The value of an argument is unchanged by the execution of the function. The term *numeric expression* indicates that the argument can be a constant, variable, or calculation.
6. 100 INPUT "Enter value";N
 100 IF SGN(N)=-1 THEN PRINT "INVALID"
 120 IF SGN(N)<>-1 THEN PRINT SQR(N)
 130 GOTO 100
7. The purpose of the "randomizing expression" is to ensure that **RND** generates an unpredictable sequence of pseudorandom numbers. Without a varying negative seed number, the same series of numbers are generated each time the **RND** function is used in a program.

8. Since **RND** cannot generate a number equal to 1, the largest number that can be generated is .99999999999999. The smallest number is zero.
9. A program to generate and sum 1000 random numbers is shown below. The program takes approximately 20 seconds to run. The sum of such a series will be close to 500 because the average value returned by **RND** is .5.

```

100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 SUM=0
120 COUNT=1
130 SUM=SUM+RND(1)
140 COUNT=COUNT+1
150 IF COUNT<1000 THEN 130
160 PRINT SUM

```

10. To generate random numbers between 100 and 199, inclusive, use the sequence $\text{INT}(100 * \text{RND}(1) + 1) + 99$. An illustrative program is shown below.

```

100 A=RND(-VAL(RIGHT$(TIME$,2))-1)
110 PRINT INT(100*RND(1)+1)+99;
120 GOTO 110

```

REVIEW TEST 4

1. (a) Valid (the number of loops depends on the values of T, A, and L)
- (b) Invalid (commas must separate data items)
- (c) Valid [number of loops = $20 + 1 - (-10) = 31$]
- (d) Valid
- (e) Valid if 3/2 is read as a string; invalid if read as a number
- (f) Invalid (data values cannot be read into calculations)
- (g) Valid (number of loops = $0 + 1 - 0 = 1$)
- (h) Valid (5.2 is rounded to 5)

- (i) Invalid (only one line number can follow **RESTORE**)
- (j) Invalid (invalid array name)
- (k) Invalid (the \$ sign is out of place)
- (l) Valid (defines only one subscripted variable)
- (m) Invalid (cannot use a subscripted variable for a *control variable*)
- (n) Valid (can use a subscripted variable for *initial value* or *limit*)
- (o) Valid
- (p) Valid
- (q) Valid [GRADE(2), GRADE2, and GRADE2! are different variables—the type declaration character differentiates GRADE2 and GRADE2!]
- (r) Invalid (cannot dimension two arrays with the same name—remember, the computer considers only the first two characters of a variable name)
- (s) Invalid (the largest line number is 65529)

2. A program to sum the whole numbers from 1 to 200, inclusive, is shown below.

```

100 SUM=0
110 FOR N=1 TO 200
120 SUM=SUM+N
130 NEXT N
140 PRINT SUM

```

3. A program that displays the results of the rich man's payment plan is shown below.

```

100 WAGES=0
110 AMOUNT=.01
120 FOR DAYS=1 TO 31
130 PRINT DAYS;AMOUNT;WAGES
140 AMOUNT=AMOUNT*2
150 WAGES=WAGES+AMOUNT
160 NEXT DAYS

```


4. Comments can be placed in a program using either the **REM** statement or the apostrophe. The apostrophe has the advantage that it can be added after any statement.
5. Dummy values are often used to mark the end of a *data list* where the number of items in the list is not fixed, but can vary from program run to program run. Select dummy values that are of the same type (numeric or string) as the data being read.
6. The **RESTORE** statement allows a program to reread a *data list*, beginning at the line number specified by the **RESTORE** statement.
7. The term *array element* is used to refer to one of the values assigned to an array variable. Each array element is identified by a unique subscript. The variable PAIR\$(5,6) names one element, although it is the element of a two-dimensional array.
8. The **DIM** statement is used to reserve memory space for an array larger than 11 elements, or to restrict the amount of memory space dedicated to an array that is less than 11 elements in size.
9. Use **DIM A(19)** to dimension the array for 20 elements.
10. The statement **DIM TEST(3,7,4)** reserves space for $4 \times 8 \times 5 = 160$ elements.
11. The function of the **RETURN** statement is to transfer execution back to the statement following the **GOSUB** statement that called the subroutine.
12. Subroutines make long programs easier to understand because they allow complicated tasks to be broken into smaller, more comprehensible tasks. This subdivision also makes long programs easier to write.

13. An **END** statement is often required when the main part of a program is followed by a subroutine. The function of the **END** statement in this situation is to prevent execution from continuing into the subroutine when execution of the main program is complete.
14. Information is passed to a subroutine by assigning it to a variable that is subsequently used by the subroutine. Information is returned to the main program in the same way, by having the subroutine assign it to a variable later used by the main program.

REVIEW TEST 5

1. (a) Invalid (a semicolon must separate the *format string* from the data being **PRINTed**)
(b) Invalid (a colon must separate CAS from the *filename*, not a semicolon)
(c) Valid (a *format string* can be a concatenated expression)
(d) Valid
(e) Valid
(f) Invalid (the colon line separator is missing)
(g) Valid ($X=X+1$ will be treated as part of a comment, not as a statement)
(h) Valid
(i) Invalid (a *format string*, if a constant, must be enclosed in quotes)
(j) Valid
(k) Valid
(l) Invalid (the **ELSE** option belongs with the **IF THEN** statement)
(m) Valid
(n) Valid
(o) Valid

- (p) Invalid (**NEXT** must be followed by the *control variable*, not the *increment*)
 - (q) Invalid (filenames must be six characters or less in length)
2. The **USING** parameter allows a display format to be specified by either a string constant or a string variable. The symbols used to specify the format are listed in Lesson 21.
 3. Multistatement lines are made by placing a colon between the statements to be combined. Multistatement lines are useful for executing multiple actions in an **IF THEN** statement, and for saving memory in long programs where memory capacity may be a limitation.
 4. The **ELSE** option in an **IF THEN** statement allows the statement to specify an action to be performed if the decision-making test is false.
 5.


```

100 FOR X=5 TO 15 STEP .5
110 Y=17*X^2+X-2
120 PRINT Y
130 NEXT X
      
```
 6.


```

100 'Read and average test scores
110 DIM GRADE(6,3)
120 'Read data loops
130 FOR O=1 TO 6
140 FOR I=1 to 3
150 READ GRADE(O,I)
160 NEXT I
170 NEXT O
180 'Average and display scores
190 FOR O=1 to 6
200 SUM=0
210 FOR I=1 to 3
      
```

```

220 SUM=SUM+GRADE(O,I)
230 NEXT I
240 AVERAGE=SUM/3
250 PRINT "Student";O;"average =";AVERAGE
260 NEXT O
270 'Test scores
280 DATA 78,82,89
290 DATA 83,85,81
300 DATA 100,70,73
310 DATA 65,68,71
320 DATA 99,89,93
330 DATA 87,0,93

```

A sample **RUN** is shown below.

```

RUN
Student 1 average = 83
Student 2 average = 83
Student 3 average = 81
Student 4 average = 68
Student 5 average = 93.6666666666667
Student 6 average = 60

```

7. (a) Turns off the internal speaker so that it does not play the sound of loading programs from cassette tape (Model 100 and 200 only)
- (b) Saves the program currently in the BASIC workspace to cassette tape with the filename CARDS.BA
- (c) Verifies that the program saved on cassette tape under the name CARDS.BA matches the program in the BASIC workspace
- (d) Load the program named MENU.BA from cassette tape

REVIEW TEST 6

1. (a) Invalid (cannot compare numeric and string values)
- (b) Invalid (the **STRING\$** argument parameters are reversed)

- (c) Invalid (cannot compare numeric and string values)
 - (d) Valid
 - (e) Valid
 - (f) Invalid (the maximum duration parameter is 250)
 - (g) Valid (the **MID\$** *length* parameter can exceed the remaining length of the string)
 - (h) Invalid (the **ASC** function requires a string argument)
 - (i) Invalid (the **VAL** function requires a string argument)
 - (j) Invalid (the maximum string length is 255 characters)
 - (k) Valid
 - (l) Valid (the *length* parameter in the **MID\$** function is optional)
 - (m) Valid
 - (n) Valid (E\$ will contain the ASCII codes to **PRINT ** ERROR** ** in "reverse video")
 - (o) Valid
2. String characters are represented inside the computer as numbers between 0 and 255. These numbers are referred to as ASCII values (ASCII is an acronym for American Standard Code for Information Interchange). When the computer compares strings for decision-making purposes, it compares the ASCII values of the characters in the strings.
3. (a) True (f) False
 (b) False (g) False
 (c) False (h) False
 (d) True (i) False
 (e) True (j) True
4. The maximum length of a string is 255 characters.
5. A program to display randomly constructed three-letter words is shown below. (Programs like this are sometimes used by companies to generate a list of potential names for new products.)
- ```

100 'Random 3-letter words
110 DIM C$(21),V$(5)
```

```

120 A=RND(-VAL(RIGHT$(TIME$,2))-1)
130 FOR COUNT=1 TO 21
140 READ C$(COUNT)
150 NEXT COUNT
160 FOR COUNT=1 TO 5
170 READ V$(COUNT)
180 NEXT COUNT
190 X=INT(21*RND(1)+1)
200 Y=INT(5*RND(1)+1)
210 Z=INT(21*RND(1)+1)
220 PRINT C$(X);V$(Y);C$(Z);" ";
230 GOTO 190 'Loop
240 DATA b,c,d,f,g,h,j,k,l,m,n,p,q,r,s,t,v,w,x,y,z
250 DATA a,e,i,o,u

```

6. A program to reverse the order of characters in a string is shown below.

```

100 'Reverse string order program
110 INPUT "Enter sample string";W$
120 REVERSE$=""
130 LCOUNT=LEN(W$)
140 FOR LOOP=LCOUNT TO 1 STEP -1
150 REVERSE$=REVERSE$+MID$(W$,LOOP,1)
160 NEXT LOOP
170 PRINT REVERSE$
180 GOTO 110

```

A sample **RUN** is shown below.

```

RUN
Enter sample string? qwerty
ytrewq
Enter sample string? MXYZTPLK
KLPTZYXM
Enter sample string? 1234567890
0987654321

```

(Press **CTRL** **C** to halt program execution.)

7. The following program converts any lowercase letters in a string value to uppercase letters.

```

100 'Convert to uppercase
110 INPUT "String value";S$
120 NS$=""
130 LCOUNT=LEN(S$)
140 FOR L=1 TO LCOUNT
150 T$=MID$(S$,L,1)
160 IF ASC(T$)<97 THEN 190
170 IF ASC(T$)>122 THEN 190
180 T$=CHR$(ASC(T$)-32)
190 NS$=NS$+T$
200 NEXT L
210 PRINT NS$
220 GOTO 110

```

The program tests each character in the string to determine if it is a lowercase letter (lines 160 and 170). If the character is a lowercase letter, 32 is subtracted from the ASCII value of the character and that new value is reconverted to a string character (line 180).

8. A program that tests for the capitals is shown below.

```

100 'Capitals of the world
110 FOR COUNT=1 TO 13
120 READ COUNTRY$,CAPITAL$
130 PRINT "Capital of ";COUNTRY$;:INPUT GUESS$
140 IF GUESS$<>CAPITAL$ THEN PRINT "Not correct. Try
 again!":GOTO 130
150 PRINT GUESS$;" is correct!"
160 NEXT COUNT
170 END
180 DATA Afghanistan,Kabul,Brazil,Brasilia, Egypt, Cairo
190 DATA England,London,Ethiopia,Addis
 Ababa,France,Paris
200 DATA Greece,Athens,India,New Delhi,Japan,Tokyo
210 DATA Mexico,Mexico City,United States,Washington
220 DATA U.S.S.R.,Moscow

```

## REVIEW TEST 7

1. (a) Valid (reserves no space for storing string data)  
 (b) Valid  
 (c) Invalid (the second comparison is not complete—it should be  $a > 14$ )  
 (d) Invalid (the **XOR** operator requires two tests, not one)  
 (e) Valid (the space between **LINE** and **INPUT** is optional)  
 (f) Valid  
 (g) Invalid (the **ON** statement requires a numeric decision value)  
 (h) Valid  
 (i) Invalid (the line is a mixture of **IF THEN** and **ON** statements)  
 (j) Valid (although pointless—this statement reserves as much string space as is already reserved)  
 (k) Invalid (there is no **LINE INPUT\$** statement)  
 (l) Invalid (the parameters of the **RIGHT\$** statement are reversed)  
 (m) Invalid (the correct statement is **INPUT\$**)  
 (n) Valid (but the order of these statements should be reversed—the **CLEAR** statement cancels the **DEFINT** statement)  
 (o) Valid (the *start position* parameter is optional)  
 (p) Valid (one **NOT** can perform the identical test: **NOT A=5**)  
 (q) Valid (the statement inputs four keystrokes)  
 (r) Invalid (the correct statement is **SPACE\$**)

2. The following program strips the leading space from converted positive numbers.

```

100 INPUT "Enter a numeric value";N
110 N$=STR$(N)
120 IF N=>0 THEN N$=RIGHT$(N$,LEN(N$)-1)
130 PRINT N$
140 GOTO 100

```

3. The **CLEAR** statement is used to reserve memory for storing string data. Because the computer automatically reserves 256



bytes of string space when BASIC is selected, the **CLEAR** statement is required only when more than 256 bytes of string space is needed.

4. The following program centers a string in a string of spaces.

```

100 INPUT "Enter a string value" S$
110 L=LEN(S$)
120 IF L>40 THEN 100
130 CENTER$=SPACE$(20-INT(L/2))
 +S$+SPACE$(20-INT(L/2+.5))
140 PRINT CENTER$
150 GOTO 100

```

The main problem to solve in writing this program is how to center strings that are an odd number of characters in length. Although you could test the length of the entered string and write separate routines for odd and even length strings, the solution shown above is more elegant. By adding .5 to the second **SPACE\$** expression, the number of spaces added to the right of an odd numbered string is reduced by 1 (adding .5 forces the expression **INT(L/2+.5)** to round up when L is odd).

5. (a) 100 PRINT LEFT\$(ANY\$,1)  
 (b) 100 PRINT RIGHT\$(ANY\$,LEN(ANY\$)-1)  
 (c) 100 PRINT RIGHT\$(ANY\$,2)  
 (d) 100 PRINT LEFT\$(ANY\$,1);RIGHT\$(ANY\$,1)  
 (e) 100 IF INSTR(ANY\$,"\" )>0 THEN PRINT "String has \"  
       ELSE PRINT "String has no \"  
 (f) 100 IF INSTR(ANY\$,"e")>0 THEN PRINT MID\$(ANY\$,  
       INSTR(ANY\$,"e"),1)  
 (g) 100 PRINT "Press any key to continue: ";A\$=INPUT\$(1)
6. The following program illustrates one method of inputting names in last name/first name order and then displaying the names in first name/last name order (without the comma).

```

100 LINE INPUT "Enter last name, first name: ";N$
110 IF N$="" THEN 100
120 P=INSTR(N$," ")
130 PRINT RIGHT$(N$,LEN(N$)-P);" ";LEFT$(N$,P-1)
140 GOTO 100

```

A sample **RUN** is shown below.

```

RUN
Enter last name, first name:
Wang,Nancy
Nancy Wang
Enter last name, first name:
Favor,Gil
Gil Favor

```

(Press **CTRL** **C** to halt program execution.)

```

7. 100 INPUT "String value";S$
 110 SS$=""
 120 FOR C=1 TO LEN(S$)
 130 T$=MID$(S$,C,1)
 140 IF T$="S" OR T$="s" THEN T$="$"
 150 SS$=SS$+T$
 160 NEXT C
 170 PRINT SS$
 180 GOTO 100

```

## REVIEW TEST 8

1. (a) Invalid (**POS** requires a dummy argument)  
 (b) Valid  
 (c) Invalid (cannot assign an error condition to a variable)  
 (d) Invalid (250 is too large to be a valid column value)  
 (e) Valid (although not advisable, an **EOF** test should precede an **INPUT #** statement)

- (f) Invalid (**CSRLIN** and **POS** return character positions, not pixel positions)
  - (g) Invalid (there is no **ON ERROR GOSUB** statement)
  - (h) Invalid (**PRESET** is a statement, not a function—therefore, it does not return a value that can be assigned)
  - (i) Valid
  - (j) Valid (as long as A\$ is a valid filename and E is a valid file number)
  - (k) Valid (the spaces are optional)
  - (l) Valid
  - (m) Valid (the # sign is optional)
  - (n) Valid (saves the program in the BASIC workspace to cassette tape in ASCII format)
  - (o) Valid
  - (p) Invalid [same reason as (h)]
  - (q) Invalid (only one transfer address is permitted)
  - (r) Invalid (the correct statement is **RESUME NEXT**)
  - (s) Invalid (**EOF** requires an argument)
  - (t) Valid (as long as F\$ is a valid filename)
  - (u) Invalid (the *on/off* parameter is missing)
  - (v) Valid (the current day setting is the filename)
2. The trailing semicolon creates a pending display state that prevents scrolling.
3. The modified screen editor program is shown below. (Lines 121–124 are new and line 160 has been changed.)

```

100 'Model 100/200 Version
110 SCREEN 0,0:CLS
120 CP=0 'Cursor Position
121 LINE (117,0)–(117,55)
122 LINE (128,0)–(128,55)
123 LINE (84,21)–(161,21)
124 LINE (84,34)–(161,34)
130 PRINT @ CP,"";

```

```

140 A$=INPUT$(1)
150 ON INSTR(1,CHR$(28)+CHR$(29)+
 CHR$(30)+CHR$(31),A$) GOSUB 180,190,200,210
160 IF INSTR("XxOo",A$)=0 THEN BEEP:GOTO 130
170 PRINT A$;:GOSUB 180:GOTO 130
180 IF CP+1>319 THEN RETURN ELSE
 CP=CP+1:RETURN
190 IF CP-1<0 THEN RETURN ELSE CP=CP-1:RETURN
200 IF CP-40<0 THEN RETURN ELSE
 CP=CP-40:RETURN
210 IF CP+40>319 THEN RETURN ELSE
 CP=CP+40:RETURN

```

```

100 'PC-8201A Version
110 SCREEN 0,0:CLS
120 C=0:R=0 'Column and Row position
121 LINE (117,0)-(117,55)
122 LINE (128,0)-(128,55)
123 LINE (84,21)-(161,21)
124 LINE (84,34)-(161,34)
130 LOCATE C,R
140 A$=INPUT$(1)
150 ON INSTR(1,CHR$(28)+CHR$(29)+CHR$(30)+
 CHR$(31),A$) GOSUB 180,200,220,230
160 IF INSTR("XxOo",A$)=0 THEN BEEP:GOTO 130
170 PRINT A$;:GOSUB 180:GOTO 130
180 IF C=39 AND R=7 THEN RETURN
190 IF C=39 THEN R=R+1:C=0:RETURN ELSE
 C=C+1:RETURN
200 IF C=0 AND R=0 THEN RETURN
210 IF C=0 THEN R=R-1:C=39:RETURN ELSE
 C=C-1:RETURN
220 IF R=0 THEN RETURN ELSE R=R-1:RETURN
230 IF R=7 THEN RETURN ELSE R=R+1:RETURN

```

4. The following program lights every other pixel. (*For the Model 200, replace the number 63 in line 110 with the number 127.*)

```

100 CLS:DEFINT C,R
110 FOR R=0 TO 63 STEP 2
120 FOR C=0 TO 239 STEP 2
130 PSET (C,R)
140 PSET (C+1,R+1)
150 NEXT C
160 NEXT R
170 GOTO 170

```

5. A data file is a collection of information stored as a unit on a computer storage device. A data record is a unit of information stored in a data file. In a sequential file, a data record always ends in a carriage return (**CHR\$(13)**). Any information that can be input into a program can be stored in a data file.
6. Explicit commas must be written between string values in the same record if you want to read the strings as separate values.
7. New information must be added at the end of a sequential file, using the APPEND mode of the **OPEN** statement.
8. A program can change the name of data file by using the **NAME** command.
9. The following program will duplicate a BASIC program that has been saved in ASCII format.

```

100 MAXFILES=2
110 ON ERROR GOTO 240
120 CLS:FILES
130 INPUT "Name of source program";S$
140 INPUT "Name for copy";C$
150 OPEN "RAM:" + S$ FOR INPUT AS 1
160 OPEN "RAM:" + C$ FOR OUTPUT AS 2
170 IF EOF(1) THEN 210
180 LINE INPUT #1,R$
190 PRINT #2,R$

```

```

200 GOTO 170
210 CLOSE 1,2
220 PRINT "Copy Done!"
230 END
240 CLOSE:BEEP
250 IF ERR=55 THEN PRINT "BAD FILE NAME OR":PRINT
 "SOURCE FILE NOT IN ASCII FORMAT":RESUME 130
260 IF ERR=57 THEN PRINT "TOO MANY FILES":END
270 PRINT "UNEXPECTED ERROR";ERR;"IN
 LINE";ERL:END

```

10. The following program enables you to input and store grade data such as that shown in the table.

```

100 CLS
110 ON ERROR GOTO 280
120 INPUT "Number of students";N
130 INPUT "Number of grades/student";G
140 OPEN "RAM:GRADES.DO" FOR OUTPUT AS 1
150 FOR C=1 TO N
160 SUM=0
170 INPUT "Student's name";N$
180 PRINT #1,N$," ";
190 FOR D=1 TO G
200 PRINT "Enter grade";D;"for ";N$;
210 INPUT GRADE
220 PRINT #1,GRADE;
230 SUM=SUM+GRADE
240 NEXT D
250 PRINT #1,USING "###.#";SUM/G
260 NEXT C
270 CLOSE:END
280 CLOSE
290 IF ERR=57 THEN PRINT "TOO MANY FILES":END
300 PRINT "UNEXPECTED ERROR";ERR;"IN
 LINE";ERL:END

```

Notice the trailing semicolons in lines 180 and 220. The semicolons create a pending print state which enables the program

to store each student's data in one record. The **PRINT #** statement in line 250 does not have a trailing semicolon, thereby ending the pending print state so that the data for the next student is stored in another record. Line 180 stores an explicit comma between the student's name and his grades, so that the name can be read back as a separate string by another program.





---

---

---

# Index

---

---

## A

- A option to **SAVE**, 273–275
- ABS** (absolute value) function, 73, 81
- Accelerated Cost Recovery System (ACRS) depreciation program, 136–137
- Addition, 11
  - multiplication before, 65–66
  - precedence level of, 68
- Addition key, 11
- Addition (+) sign, 29–30
  - for printing numeric data, 145
- ADDRESS program, 5
- Addresses, return, 134
- American Standard Code for Information Interchange (*see* ASCII *entries*)
- AND** operator, 215–216
- Apostrophes indicating remarks, 110
- APPEND mode, 259, 266
- Arguments, 72
- Arithmetic, modulus, 68
- Array name, 124
- Array size, 124
- Array variables, 120–131
- Arrays, 120–131
  - dimensioning, 125
  - numeric, 122
  - one-dimensional, 127
  - two-dimensional, 127–128

**ASC** (arccosine) function, 192–196  
 ASCII code functions, 192–197  
 ASCII codes and characters, 285–294  
 ASCII format, programs in, 273–274  
 ASCII sequence, 295–296  
 ASCII values, 178–180, 197  
   identifying row and column positions, 297–298  
   printing, 296–297  
 Assignment of values, 17–19  
 Asterisk key, 12  
 Asterisks (\*), 60  
   double, 146  
**ATN** (arctangent) function, 73

## B

“Backing up,” 166  
 Backslash (\) character, 67  
   for printing string data, 145  
 Backspace key, 9–10  
 Bad subscript message, 124, 129  
 BASIC, Microsoft, 5–6  
 BASIC reserved words (*see* Reserved words)  
 Batteries, 4  
**BEEP** statement, 172–173, 194  
*box/fill* parameter, 252  
 Built-in software, 5

## C

Calculations, order of, 65–71  
   (*See also* Levels of precedence)  
 Calling subroutines, 134  
 Card program, 137–139  
 Caret (^), 67, 146  
 Carriage return, 197

Carriage return key, 3  
 Cassette tape, 165–167  
   storing programs on, 165–169  
 Chance occurrences, simulating, 89–96  
 Character spaces, 10–11  
**CHR\$** function, 55, 193–196  
**CLEAR** statement, 205  
**CLOAD?** command, 167–169  
 Clock, electronic, 5  
**CLOSE** statement, 257–258, 263–264  
**CLS** statement, 14  
 CMOS (complementary metal-oxide semiconductor), 4  
 Colons (:) separating multiple statements, 151–155  
 Column size, 128  
 Command mode, 21  
 Commands, 22  
 Commas, 13–14, 269  
   explicit, 260–261  
   separating input variables with, 41  
 Comparisons, string, 176–182  
 Complementary metal-oxide semiconductor (CMOS), 4  
 Computer messages (*see* Messages)  
 Computers, portable, 1–7  
 Concatenation of strings, 29–30  
 Constants:  
   numeric, 16  
   string, 28  
 Control variables, 99–106, 160–162  
**COS** (cosine) function, 73  
**CSAVE** command, 166–167  
**CSRLIN** function, 246  
 CTRL key, 32  
**CTRL C** keys, 44

Current files, 60  
 Cursor, flashing, 9  
 Cursor movement keys, 3

## D

Data, storing, in programs, 111–119  
 Data files, 257–272  
   relative, 265  
   sequential (*see* Sequential data files)  
 Data list, 112, 113  
**DATA** statement, 112–119, 257  
**DATES**, 275–276  
**DAYS**, 275–276  
 Decimal codes, 285–294  
 Decision making, 47–52, 156–158  
 “Default” instructions, 54  
 DEFDBL letter range, 86  
 DEFINT letter range, 86  
 DEFSNG letter range, 86  
 DEFSTR letter range, 86  
 Deleting files, 60  
 Denominators, evaluating, 69  
 Depreciation program, 136–137  
 Device names, 259  
**DIM** statement, 124–129  
 Dimensioning arrays, 125  
 Display messages (*see* Messages)  
 Display screen (*see* Screen, display)  
 Division, 12  
   integer, 67–68  
   precedence level of, 67  
 Dollar sign (\$), 28, 29, 84–85  
   double, 146  
 Dot-addressable graphics, 4  
 Double precision, 83–86  
 Doubly dimensioned array message, 125

DOWN key, 32  
 “Dummy” values, 116

## E

**EDIT** command, 31–35  
 Editor, screen, 242  
 8085 microprocessor, 4  
 Electronic clock, 5  
 Elements, 120–121  
**ELSE** option, 156–158  
**END** statement, 23, 134  
 Enter key, 3, 22  
   function of, 10  
**EOF** function, 258, 262–263  
 Equal-to (=) symbol, 48, 178  
 Equals-or-follows-in-ASCII-order  
   (>=) symbol, 178  
 Equals-or-precedes-in-ASCII-order  
   (<=) symbol, 178  
 Equals (=) sign, 16–19  
**EQV** (equivalence) operator, 217  
 Erasing mistakes, 9  
**ERL** function, 278  
**ERR** function, 278  
 Error code(s), 278  
   list of, 301–303  
 Error handling, 277–280  
 Error messages (*see* Messages)  
**ERROR** statement, 278–280  
 Errors, syntax, 32  
 Exclamation point (!), 84–85  
   for printing string data, 145  
 Exclusive **OR (XOR)** operator, 216–217  
**EXP** (natural antilogarithm) function, 73  
 Explicit commas, 260–261  
 Exponential notation, 199  
 Exponentiation, 66–67

Expression list, 13–14  
 semicolons and, 144

Expressions:  
 numeric, 68–70, 72–74, 211  
 string, 30

Extension, file, 58

## F

**F10** function key, 14

Factorials, 103–104

File extension, 58

File limit error, 59

File number, 259

Filenames, 259

Files, 57

current, 60

data (*see* Data files)

deleting, 60

RAM, 57–61

resident, 60

**FIX** function, 76–79

Flashing cursor, 9

Follows-in-ASCII-order ( $\succ$ ) symbol,  
 178

**FOR TO NEXT** statement, 159–  
 164

looping with, 99–107, 159–164

nesting, 159–164

Format string, 144–149

**FRE** function, 205–207

Function call, illegal, 59, 184, 187,  
 223

Function keys, 3, 53–56

Functions:

ASCII code, 192–197

higher-order mathematical, 66,  
 72–74

trigonometric, 73–74

## G

Global declaration statement, 86–87

**GOSUB** statement, 132–140

**GOTO** statement, 43–46

implied, 51

**GRAPH** - keys, 67

Graphics:

dot-addressable, 4

statements for, 251–256

Greater than ( $\succ$ ) symbol, 48

Greater-than-or-equal-to ( $\succ=$ ) sym-  
 bol, 48

Gregorian calendar program, 225–  
 228

## H

Hexadecimal codes, 285–294

Higher-order mathematical func-  
 tions, 66, 72–74

Hyphen (*see* Minus sign)

## I

**IF THEN** statement, 47–52, 102, 156

**IF THEN ELSE** combination, 156–  
 158

Immediate execution mode, 21

**IMP** (implication) operator, 217–218

Implied **GOTO**, 51

Increments, 162–163

Infinite loops, 44

Initial values, 99–106

as zero, 18

**INKEY\$** function, 212–213

**INPUT** statement, 38–42, 209

**INPUT #** statement, 258, 261–262

**INPUT\$** function, 211–212, 258, 262

INPUT mode, 259, 266  
 Input/output ports, 4–5  
**INSTR** function, 231–233  
**INT** function, 75–78  
 Integer division, 67–68  
 Integer precision, 83–86  
 Integers, 75

## K

**KEY** command, 55  
**KEY LIST** command, 54  
 Key number, 55  
 Key sequence, 285–294  
 Keyboard buffer, 3  
 Keyboards, 3  
 Keys, function, 3, 53–56  
 Keywords, 19  
**KILL** command, 60, 258, 263–264

## L

Label key, 54  
 LCD (liquid crystal display), 4  
**LCOPY** instruction, 276  
 Leading spaces, 13  
 Left arrow ( $\leftarrow$ ) symbol, 32  
**LEFT\$** function, 229  
**LEN** function, 186–187, 190  
 Length parameter, 172–173  
 Less than ( $<$ ) symbol, 48  
 Less-than-or-equal-to ( $\leq$ ) symbol, 48  
**LET** statement, 18–19  
 Letter-range parameters, 86  
 Letters, uppercase or lowercase, 11  
 Levels of precedence:  
   for logical operators, 219  
   for mathematical operations, 66–69

Limits, 100–106  
**LINE** statement, 251–254  
 Line end ( $<$ ) symbol, 32  
**LINE INPUT** statement, 210–211  
**LINE INPUT #** statement, 258, 262  
 Line-number range, 23–24  
 Line-number-range parameter, 34  
 Line numbers, 21–25  
 Liquid crystal display (LCD), 4  
**LIST** command, 23–25  
 Literals, 148  
**LLIST** instruction, 276  
**LOAD** command, 59  
**LOCATE** statement, 237–243, 244–245  
**LOG** (natural logarithm) function, 73  
 Logical operations, 215–222  
   levels of precedence for, 219  
 Loops:  
   controlling, 49–51  
   **FOR TO NEXT**, 99–107, 159–164  
   infinite, 44  
 Low-battery indicator, 4  
 Lowercase letters, 11  
**LPOS** instruction, 277  
**LPRINT** instruction, 276  
**LPRINT USING** instruction, 277

## M

Mathematical functions, higher-order, 66, 72–74  
 Mathematical operations, 65–71  
**MAXFILES** instruction, 258–260  
**MENU** command, 14  
 Menu screen, 8–9  
 Merging programs, 273–275

Messages, 10

- BN Error, 260
- Break in XXX, 44
- BS Error, 124, 129
- DD Error, 125
- error, 10
- FC Error, 59, 184, 187, 223
- FL Error, 59
- ID Error, 261
- list of, 301-303
- OD Error, 115, 278-279
- Ok, 8-9
- OS Error, 30, 204, 205
- OV Error, 67, 85
- Redo from start, 39
- RG Error, 134
- SN Error, 32, 70
- TM Error, 28, 198
- UL Error, 26
- Verify failed, 167
- Microprocessor 8085, 4
- Microsoft BASIC, 5-6
- MID\$** function, 183-185, 193
- Minus sign (-), 12, 19
  - for printing numeric data, 145-146
- Mistakes, erasing, 9
- MOD** operation, 68
- Mode parameter, 259
- Models 200 and 100, 1-7
  - ASCII codes and characters, 285-294
  - error codes and messages, 301-302
  - Menu Screen of, 9
  - PRINT @** values used by, 238
  - reserved words on, 299-300
- Modems, 1, 3

- Modulus arithmetic, 68
- Multiple statements, 151-155
- Multiplication, 12
  - before addition, 65-66
  - precedence level of, 67

N

- NAME** command, 59-60
- Names, variable (*see* Variable names)
- NEC PC-8201A (*see* PC-8201A)
- Negative numbers, 12-13
- Negative of, taking the, 67
- Nesting:
  - FOR TO NEXT** loops, 159-164
    - subroutines, 134-136
- NEW** command, 18, 23, 32, 57, 207
- NEXT** statement, 99-107, 159-164
- Nickel-cadmium (nicad) battery, 4
- Not-equal-to (<>) symbol, 48, 178
- NOT** operator, 218
- Null strings, 29, 180
- Numbers:
  - converting strings to, 198-200
  - line, 21-25
  - negative, 12-13
  - positive, 12-13
  - pseudorandom, 89-96
- Number symbol (#), 84-85
  - number of, 147
  - for printing numeric data, 145
- Numerators, evaluating, 69
- Numeric arrays, 122
- Numeric constants, 16
- Numeric data, characters for printing, 145-146
- Numeric expressions, 68-70, 72-74, 211
- Numeric variables, 16-20, 83-88

## O

**ON ERROR GOTO** statement, 277  
**ON GOSUB** statement, 223–228  
**ON GOTO** statement, 223–228  
*on/off* parameter, 54, 252  
 One-dimensional arrays, 127  
**OPEN** statement, 257–263  
**OR** operator, 216  
   exclusive (**XOR**), 216–217  
 Order of calculations, 65–71  
   (*See also* Levels of precedence)  
 Out of data error, 115, 278–279  
 Out of string space message, 30,  
   204, 205  
 OUTPUT mode, 259, 266  
 Overflow error, 67, 85

## P

P, value of, 78–79  
 Parentheses, 69–70  
   matching pairs of, 70  
 PC-8201A, 1–7  
   ASCII codes and characters, 285–  
     294  
   error codes and messages, 302–  
     303  
   **LOCATE** row and column values,  
     238  
   Menu Screen of, 9  
   reserved words on, 299–300  
 Pending print state, 45  
 Percent sign (%), 84–85  
 Piezoelectric tone generator, 5  
 Pitch parameter, 172–173  
 Pixels, 251–252  
 Plus sign (*see* Addition sign)  
 Portable computers, 1–7  
**POS** function, 245–246

Positive numbers, 12–13  
 Power switch, 4  
 Powers, raising to, 66–67  
 Precedence, levels of (*see* Levels  
   of precedence)  
 Precedes-in-ASCII-order (<) sym-  
   bol, 178  
 Precision, degree of, 83–86  
**PRESET** statement, 254  
**PRINT** statement, 10–13  
**PRINT @** statement, 237–244  
**PRINT #** instruction, 258, 260–261  
 Print positions, controlling, 237–250  
 Print state, pending, 45  
**PRINT USING** statement, 144–150  
**PRINT # USING** instruction, 258,  
   261  
 Print values, 13–14  
 Print zones, 13–14  
 Printer instructions, 276–277  
 Printing:  
   numeric data, 145–146  
   string data, 145  
 Programming, 21–22  
 Programs:  
   adding sound to, 172–175  
   in ASCII format, 273–274  
   card, 137–139  
   depreciation, 136–137  
   merging, 273–275  
   optimizing, for space and speed,  
     280–281  
   storing, on cassette tape, 165–  
     169  
   storing data in, 111–119  
   user-friendly, 247–248  
 Prompt parameter, 210  
 Prompts, question mark, 38–40  
**PSET** statement, 254–255  
 Pseudorandom numbers, 89–96  
 Punctuation keys, 3

## Q

Question mark(s) (?), 11–12, 19  
     as prompt, 38–40  
     two, 42  
 Quotation marks ("), 27–30  
     strings and, 112  
 Quotient, truncated, 67–68

## R

Radians, 74  
 Radio Shack TRS-80 Model 100  
     (see Model 100)  
 Radio Shack TRS-80 Model 200  
     (see Model 200)  
 Raising to powers, 66–67  
 Random access memory (RAM), 1  
     files, 57–61  
 Random occurrences, simulating,  
     89–96  
 Read-only memory (ROM), 5  
**READ** statement, 113–117  
 Reading sequential files, 267–269  
 Relational tests, 47–52  
 Relative data files, 265  
**REM** statement, 108–110  
 Remainders, 68  
 Remarks, apostrophes indicating,  
     110  
 Remote motor control, 166  
 Repeating keys, 3  
 Reserved words, 19  
     list of, 299–300  
 Reserving string space, 204–208  
 Resident files, 60  
**RESTORE** statement, 117–118  
**RESUME** statement, 277–278  
 Return addresses, 134  
**RETURN** statement, 133–139

Return with **GOSUB** message, 134  
 Returning values, 73  
 Reverse video, 197  
 Review Test 1, 36–37  
     answers to, 304–306  
 Review Test 2, 62–64  
     answers to, 306–309  
 Review Test 3, 97–98  
     answers to, 309–311  
 Review Test 4, 141–143  
     answers to, 311–314  
 Review Test 5, 170–171  
     answers to, 314–316  
 Review Test 6, 201–203  
     answers to, 316–319  
 Review Test 7, 235–236  
     answers to, 320–322  
 Review Test 8, 282–284  
     answers to, 322–327  
 Right arrow (→) key, 32  
**RIGHT\$** function, 229–230  
**RND** function, 89–96  
 ROM (read-only memory), 5  
 Rounding values, 78–79  
 Row size, 128  
**RUN** command, 21–23

## S

**SAVE** command, 57–59, 166–167  
     A option to, 273–275  
 SCHEDL program, 5  
 Scientific notation, 199  
 Screen, display, 4  
     editor for, 242  
     menu on, 8–9  
     messages on (see Messages)  
     scrolling process, 12, 241  
**SCREEN** command, 54  
**SCREEN** 0,0 instruction, 246



- Scrolling, 12
  - preventing, 241
- Seed values, 90-91
- Semicolons (;), 13-14, 40, 44-45, 267
  - expression lists and, 144
  - trailing, 241
- Sequential data files, 265-272
  - reading, 267-269
  - writing, 268-271
- SGN** function, 80-81
- SHIFT 6**, 67
- SHIFT BREAK** keys, 39
- Shift key, 3, 14
- Simple variables, 120
- Simulating chance occurrences, 89-96
- SIN** (sine) function, 73
- Single precision, 83-86
- Slash (/), 19
- Slash key, 12
- Software, built-in, 5
- Sound:
  - adding, to programs, 172-175
  - capabilities for, 5
- SOUND** statement, 172-175
- SOUND ON/OFF**, 168, 173
- Space, memory, optimizing programs for, 280-281
- SPACES** function, 230-231
- Spaces:
  - ASCII value of, 179
  - character, 10-11
  - leading, 13
  - strings and, 28-29
  - trailing, 13
- Speed, optimizing programs for, 280-281
- SQR** (square root) function, 73
- Statements, 22
  - for graphics, 251-256
  - multiple, 151-155
- STOP** key, 39
- Storing:
  - data in programs, 111-119
  - programs on cassette tape, 165-169
- STR\$** function, 199-200
- String constants, 28
- String expressions, 30
- String variables, 28
- STRING\$** function, 187-188, 190-191
- Strings, 27-30
  - characters for printing, 145
  - comparing, 176-182
  - concatenation of, 29-30
  - converting numbers to, 198-200
  - format, 144-149
  - length of, 29-30
  - manipulations of, 182-191
  - null, 29, 180
  - quotation marks and, 112
  - reserving space for, 204-208
  - spaces and, 28-29
- Subroutines, 132-140
  - calling, 134
  - nesting, 134-136
- Subscripts, 120-131
- Subtraction, precedence level of, 68
- Syntax errors, 32, 70
  
- T**
  
- TAN** (tangent) function, 73
- Tandy TRS-80 Model 100 (see Model 100)

Tandy TRS-80 Model 200 (*see* Model 200)  
 TELCOM program, 5  
 TEXT program, 5  
**THEN** portion of **IF THEN** statement, 47–52  
**TIMES** statement, 91–92, 275–276  
 Tone generator, piezoelectric, 5  
 Trailing spaces, 13  
 Transfer address, 43  
 Trigonometric functions, 73–74  
 TRS-80 Model 100 (*see* Model 100)  
 TRS-80 Model 200 (*see* Model 200)  
 True-false tests, 215–216  
 Truncated quotient, 67–68  
 Two-dimensional arrays, 127–128  
 Type declaration character, 84  
 Type mismatch error, 28, 198

## U

Unary operations, 67  
 Undefined line message, 26  
 Uppercase letters, 11  
 User-friendly programs, 247–248  
**USING** parameter, 144–150

## V

**VAL** function, 198–200  
 Values:  
     ASCII (*see* ASCII values)  
     assignment of, 17–19

“dummy,” 116  
 initial (*see* Initial values)  
 print, 13–14  
 returning, 73  
 rounding, 78–79  
 seed, 90–91

Variable list, 113  
 Variable names, 18–19  
     first two characters of, 19  
 Variables:  
     array, 120–131  
     control, 99–106, 160–162  
     numeric, 16–20, 83–88  
     simple, 120  
     string, 28  
 Video, reverse, 197

## W

Words, reserved (*see* Reserved words)  
 Writing sequential files, 268–271

## X

**XOR** (exclusive **OR**) operator, 216–217

## Z

Zero, initial values as, 18  
 Zones, print, 13–14

## Catalog

If you are interested in a list of fine Paperback books, covering a wide range of subjects and interests, send your name and address, requesting your free catalog, to:

McGraw-Hill Paperbacks  
1221 Avenue of Americas  
New York, N. Y. 10020

# PORTABLE BASIC

**Programming the TRS-80 Model 200,  
TRS-80 Model 100, and NEC PC-8201A Computers**

---

**DAVID THOMAS**

This book is all you'll need to program your TRS-80 Model 200, TRS-80 Model 100, or NEC PC-8201A computer in the simple-to-learn BASIC programming language. PORTABLE BASIC is organized into 40 short, easy-to-follow lessons. Each lesson covers an important programming concept or a BASIC statement or command. Review exercises, designed to reinforce what you've learned, follow every five lessons.

- Designed for people who want to learn BASIC fast and have no previous computer experience.
- Uses a practical, hands-on approach to BASIC, the most popular of computer programming languages.
- Allows you to begin learning immediately, as it contains no lengthy technical or theoretical discussions.
- Covers all the major statements and commands of Model 200, Model 100, and PC-8201A BASIC.
- Contains many example programs that can be used by engineering and scientific professionals, students, business people, managers, hobbyists, and computer enthusiasts.
- Includes complete answers to all review exercises.
- Describes computer capabilities not documented in the manuals supplied with the computers.

A lesson-oriented book, PORTABLE BASIC provides excellent examples, problems, review exercises, and user interaction that allows you to *learn by doing*.

**David Thomas** is a freelance writer and consultant to Radio Shack and Texas Instruments with many years of experience in writing about programming products and computers. He is the president of Innovative Services, Inc., and is the author of *Learn BASIC: A Guide to Programming the Texas Instruments Compact Computer 40*, a BYTE/McGraw-Hill book.



ISBN 0-07-064260-5